

Automated Analysis of Nash Equilibrium in Two-Player Games



Tong Gao
Worcester College
University of Oxford

Supervised by Dr. Julian Gutierrez

A thesis submitted for the degree of
Msc in Computer Science

Trinity 2016

Acknowledgements

I would like to thank my supervisor, Dr. Julian Gutierrez for his outstanding support and sincere commitment on helping and guiding me throughout the project. A further gratitude to Professor Alessio R. Lomuscio for his support on the MCMAS model checking tool.

Abstract

Nash Equilibrium is one of the fundamental concepts in game theory. This project aims to solve the ‘Existence of Nash Equilibria’ problem in two-player concurrent iterated Boolean games. An explicit algorithm is used, by reducing the current problem into a series of satisfiability and synthesis problems using the temporal logics LTL (Linear-time Temporal Logic) and CTL* (Computational Tree Logic). In this project, we implement the decision procedure for checking the existence of Nash equilibria by using MCMAS, a powerful model checking tool for concurrent and multi-agent systems. Python is used as the implementation language. The program has been proved to achieve excellent performance for small specification, where for larger systems our results show that better algorithms for synthesis, as offered by MCMAS, may be needed.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Objectives	2
1.3	Dissertation Structure	3
2	Background	5
2.1	Iterated Boolean Game	5
2.2	Nash Equilibrium	6
2.3	Temporal Logic	7
2.3.1	LTL (Linear-time Temporal Logic)	7
2.3.2	CTL* (Computation Tree Logic*)	8
2.3.3	SL (Strategy Logic)	9
2.4	Satisfiability	10
2.5	Synthesis	11
3	Algorithms	13
3.1	Overview	13
3.2	Problem Definition	13
3.3	The Game	14
3.4	The Algorithm	16
4	Design Implementation	21
4.1	Overview	21
4.2	Introduction to MCMAS	21
4.2.1	What is MCMAS ?	21
4.2.2	Model Checker: MCMAS-SL	24
4.2.3	ISPL File	25
4.2.4	MCMAS Simulation	26
4.3	Algorithm Implementation	27

4.3.1	Read User Input File	28
4.3.2	Translate to ISPL File	29
4.3.2.1	Agents	29
4.3.2.2	Evaluation	32
4.3.2.3	Initstates	33
4.3.2.4	Formulae	33
4.3.3	Run MCMAS	36
4.3.4	Parse the Output	37
4.4	Design Extensions	38
4.4.1	Explore: ALL vs WHICH	38
4.4.2	The ‘X’ Operator	39
5	System Evaluation	41
5.1	Overview	41
5.2	System Testing	41
5.3	Performance Evaluation	46
5.3.1	Case Study 1: Simple goals, vary the number of variables	46
5.3.2	Case Study 2: Few variables, vary the complexity of goals	47
5.3.2.1	Goals with F only	47
5.3.2.2	Goals with G only	48
5.3.2.3	Goals with U only	50
5.3.2.4	Goal with G F U X	51
5.3.2.5	Summary	52
5.3.3	Case Study 3 : ALL vs WHICH	52
6	Conclusion	55
6.1	Dissertation Remarks	55
6.2	Achievements	56
6.3	Recommendations for Future Work	57
A	Python Source Code	59
A.1	sourcode.py	59
A.2	main.py	71
A.3	test.py	72
A.4	sample input.ibg	77
A.5	sample input.ispl	78

B User Instructions	81
B.1 Set Up	81
B.2 Run MCMAS	81
B.3 Result Collection	82
Bibliography	83

List of Figures

1.1	Techniques used in solving the existence of NE problem	3
3.1	Actions performed by the players	15
3.2	Goal Evaluation	15
3.3	Strategy Selection Process	18
3.4	Working Principle of Winning Strategies for Player 1 and 2	19
3.5	Working Principle of Winning Strategy σ_A	20
4.1	Implementation Structure of MCMAS	22
4.2	Sample Agent defined in ISPL code	23
4.3	Sample ISPL file	25
4.4	MCMAS Output	26
4.5	System-level Design	27
4.6	User Input File	28
4.7	Agent Environment	29
4.8	Agent Player with one variable	30
4.9	Agent Player with two variables	30
4.10	Agent Player with three variables	30
4.11	Function <i>createPlayer()</i>	31
4.12	Evaluation coded in ISPL	32
4.13	Initstates coded in ISPL	33
4.14	Formulae coded in ISPL	35
4.15	Function <i>runMCMAS()</i>	36
4.16	Output Parser	37
5.1	MCMAS output with option ‘ALL’	42
5.2	MCMAS output with option ‘WHICH’	43
5.3	MCMAS output for the Matching Pennies Game	45
5.4	Execution Time with option ‘ALL’	52
5.5	Execution Time with option ‘WHICH’	53

List of Tables

5.1	Goal with F only: Two variables	47
5.2	Goal with F only: Three variables	48
5.3	Goal with G only: Two variables	48
5.4	Goal with G only: Three variables	49
5.5	Goal with G only: Four variables	49
5.6	Goal with U only: Two variables	50
5.7	Goal with U only: Three variables	50
5.8	Goal expressed in full LTL: Two variables	51
5.9	Goal expressed in full LTL: Three variables	51

Chapter 1

Introduction

1.1 Motivation

Game theory refers to the study of the rational responses within individual or groups of players under specific conditions, where each player's action depends on other players' choices [1]. Nash equilibrium, as one of the fundamental concepts in game theory, is widely used to make predictions on the rational decisions made by players, where interactions are involved. Informally, we say that a game reaches a Nash equilibrium when no player has an incentive to deviate from his or her chosen strategy, while taking into consideration the opponents' choices [2].

In the past few years, several techniques have been used to solve the 'Existence of Nash equilibria' problem. The automata-theoretic approach, for instance, is one of the most popular techniques used nowadays [3]. This approach was originally used to solve the rational synthesis problem, which can be reduced into solving the emptiness problem of an alternating parity automata on trees, a problem that is known to be very hard to solve in practice. Besides, the problem can also be solved with computational logic, by using FSM (Finite State Machine) to represent the strategies of each player [4]. However, no matter what techniques we use, the existence of Nash equilibria is regarded as one of the computationally hardest problems in (algorithmic) game theory, and in particular in the games approach to the analysis in multi-agent systems. Thus, researchers are eager to find simpler and more elegant ways to solve this problem.

In a recent paper written by Julian Gutierrez, Paul Harrenstein, and Michael Wooldridge, it has been proved that for games with two players and goals succinctly given by formulas of Linear Temporal Logic (LTL), the 'Existence of Nash Equilibria' problem can be converted into simpler sub-problems using finite-memory strategies [5]. An explicit algorithm for solv-

ing this problem is defined in the two-player game. However, currently the algorithm only exists in theory, but not a practical implementation has been investigated. Thus, the aim of this project is to implement the decision procedure for checking the existence of Nash equilibria in two-player iterated Boolean games in practice, and to evaluate the performance of the system.

1.2 Objectives

The existence of Nash equilibria can be understood as the problem of checking whether there is a profile of strategies, one for each player in the game, so that no player has a beneficial deviation, that is, an incentive to change to a different or alternative strategy. Figure 1.1 presents three different techniques that can be used to solve this particular problem in iterated Boolean games. Notice that the first two techniques can be used for the n-player setting, while the last technique can only be used in the two-player setting.

In this project, we focus on the use of temporal logics for two-player iterated Boolean games. The algorithm uses temporal logic-based techniques by converting the current problem into a number of satisfiability and synthesis problems [5]. The question we want to answer is whether checking the existence of Nash equilibria can be solved efficiently in practice when we consider games with two players. That being said, the objectives of this project are defined as follows:

- Learning about the formal verification of concurrent and multi-agent systems using games, temporal logic, and model checking techniques.
- Constructing an iterated boolean game based on information given by the users.
- Implementing the algorithm to check the existence of Nash equilibria in two-player games by solving a number of LTL (Linear-time Temporal Logic) satisfiability and CTL* (Computation Tree Logic*) synthesis problems using MCMAS, a model checking tool used for multi-agent systems.
- Developing the program so that it is flexible enough to adapt different types of iterated Boolean games.
- Exploring the performance of the system by conducting several experiments.

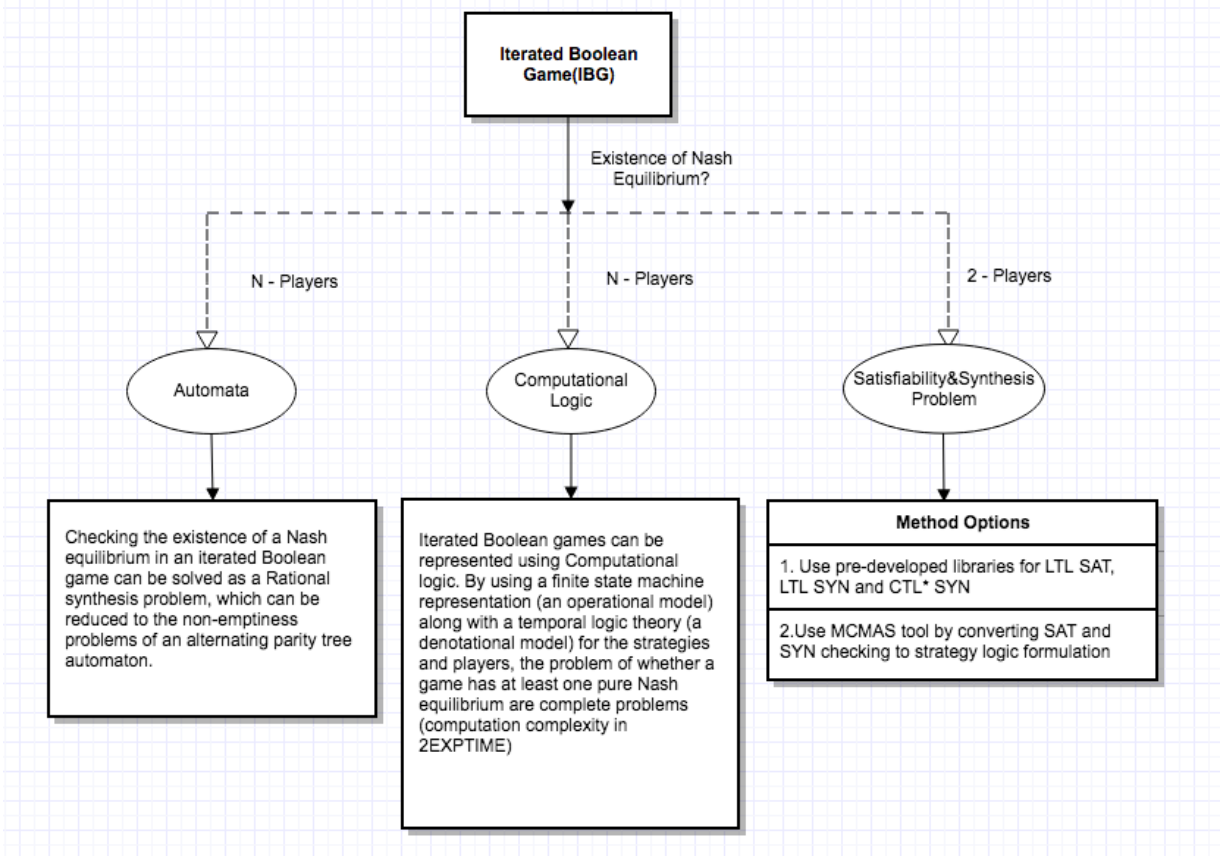


Figure 1.1: Techniques used in solving the existence of NE problem

1.3 Dissertation Structure

This dissertation is divided into six chapters. Chapter 1 gives an introduction to the project, including project motivation, problem statement, project objectives, as well as the overall structure of the dissertation.

Chapter 2 provides the essential background information behind the project. This chapter introduces the concept of iterated Boolean games, and how players behave in such a game. We will also talk about what Nash equilibrium is, and how to achieve Nash equilibrium. Besides, several temporal logics will be introduced, including LTL (Linear-time Temporal Logic), CTL* (Computation Tree Logic*) and SL (Strategy Logic). Last, we will talk about the satisfiability and synthesis problems for temporal logics.

Chapter 3 presents the algorithm used in the implementation. This chapter defines the specific problem that needs to solve, and describes the setting and structure of the game. Finally, the explicit algorithm will be explained in details.

Chapter 4 describes the main implementation of the system. First, MCMAS is introduced, which is the tool used to perform satisfiability and synthesis checking. Then, we will talk about implementation details, including both system-level design and block-level design, and two design extensions.

Chapter 5 demonstrates the usability and evaluates the performance of the system. Two benchmarks are presented for system testing, and several case studies are carried out for performance evaluation.

Chapter 6 concludes the whole dissertation. In this chapter, we give a quick recap on the project, and present the results and findings of our work. Several open questions will be brought up, and future work will be discussed.

Chapter 2

Background

2.1 Iterated Boolean Game

One-shot games and iterated games are two well-known games in the game theory literature. Compare to the one-shot game, where the players can only play one time, iterated game allows the players to play repeatedly after they see other players' choices in the past [4]. This leads to the term of 'rational cooperation', where the players have the opportunity to cooperate with others to achieve a better result, or to punish other players for non-cooperation or betrayal.

This project focuses on the framework of iterated Boolean games, which is a concrete computational model used to explore questions in rational verification [6]. An iterated Boolean games is 'played' over an infinite sequence of rounds, where in each round, the players select a valuation for the set of boolean variables they control [6]. The goal of each player is expressed in temporal logic (LTL). For instance, player p_1 is associated with goal g_1 , and controls a set of Boolean variables $V = \{v_1, v_2, v_3 \dots\}$, where $V(n) = 2^n$. This means for n variables, there are 2^n assignments. For each round, p_1 will assign values to the set of variables, by selecting a subset of variables V' where $V' \subset V$. All variables in V' are assigned to true, while others are assigned to false. One unique feature of iterated Boolean games is that whether the player's goal is satisfied is depend on how he and other players act in the future rounds. Thus, rational strategies are needed. Strategies allow the players to watch, investigate and analyze others' behaviour, and give a response that will maximize his or her probability to achieve the goal, or minimize the probability of others to achieve their goals.

The complexity of various decision problems for iterated Boolean games has been studied. Model checking is PSPACE-complete, synthesis is 2EXPTIME-complete, and checking whether an iterated Boolean game has a Nash equilibrium is 2EXPTIME-complete [4].

2.2 Nash Equilibrium

Nash equilibrium is the most common and important solution concept in game theory. It was first brought up by John F. Nash in the paper “Equilibrium Points in n-Person Games” in 1950 [7]. In this paper, John F. Nash used mathematical to prove the following theorem: ‘For any n-player zero-sum or non-zero-sum game, if each player has a finite set of strategies, then there must be at least one equilibrium point’. In 1994, John F. Nash won the Nobel Memorial Prize in Economic Sciences, and this paper has become one of the landmarks of the game theory literature.

We say that a game reaches a Nash equilibrium when no player has an incentive to deviate from his or her chosen strategy after considering the opponents’ choices [8]. There can be more than one Nash equilibrium point in a single game. The Prisoner’s Dilemma is a standard example of a non-zero-sum game in game theory [9]. The game is presented as follows: Two criminals are arrested and under interrogations separately without the chance to communicate with each other. If both of them cooperate with each other and keep silence, they will be jailed for only one year due to the lack of evidence. If one of them betrays and reports on the other person’s crime, while the other person remains silent, then the one who reported will be acquitted of charges, while the other will be jailed for five years. Last, if both of them betray each other, then both will be jailed for 3 years. In this game, there’s only one possible Nash equilibrium point, which is ‘both criminals betray each other’. To give a simple explanation, let’s think from criminal A’s point of view. Suppose A remains silence and cooperate with B. If B remains silence, A will be jailed for one year. If B betrays, A will be jailed for 5 years. Suppose A betrays B and report B’s crime. If B remains silence, A will be free to go. If B betrays, A will be jailed for 3 years. We can easily see that no matter what B acts, A betrays B will maximize A’s own benefits and vice versa. Thus, we can reach the conclusion that Nash equilibrium forms when both criminals defect.

In reality, the theory of Nash equilibrium has made a huge impact on Economic Science. It has changed the system and structure of Economics. It also enlarges and strengthens the relationship between Economics and other subject areas that are closely related, such as social science and natural science. The greatness about Nash equilibrium is that it is so common and exists almost everywhere. Thus, studying Nash equilibrium has become an important topic in game theory among researchers and scientists. In this project, we only consider pure strategy Nash equilibrium.

2.3 Temporal Logic

In logic, the term temporal logic is used to describe the systems or representations that reasoning about time [10]. In simple terms, temporal logic is one kind of logics with temporal operators, such as next, eventually, always, until and etc. Thus, it is also called tense logic in some literature. Temporal logic is mainly used in formal verification, to define or state the requirements of a system or program. In game theory, one can use temporal logic to construct and formalize games. In this project, we focus on three types of temporal logic, Linear-time Temporal Logic (LTL) [11], Computation Tree Logic* (CTL*) [12] and Strategy Logic (SL) [13].

2.3.1 LTL (Linear-time Temporal Logic)

LTL stands for Linear-time Temporal Logic, which is designed to specify the temporal relations between systems. It can be seen as propositional logics augmented by temporal operators that specify the properties of reactive systems [14]. In LTL, one can write formulae to represent the events that will happen in the future.

Syntax LTL formula is composed of a finite set of atomic propositions (AP), basic logical operators (\neg for negation and \wedge for conjunction), and temporal operators. It can also have some additional logical operators, such as \vee (disjunction), \rightarrow (implication) and \leftrightarrow (equivalence). The temporal operators can be divided into two types [11] :

Unary operators:

- X = next : indicates the current statement has to hold at the **next** time step
- G = always : indicates the current statement has to hold **all** the time in the future
- F = eventually: indicates the current statement has to hold **sometimes** in the future

Binary operators:

- R = release : formula ' $\gamma R \theta$ ' states that θ has to hold until γ holds
- U = until : formula ' $\gamma U \theta$ ' states that γ has to hold until θ holds

To give an example, consider the following LTL formula: ' $G F \neg x$ ', with two unary operators G and F. Here, the formula states that 'it is always true that x will be false sometimes in the future'.

In this project, LTL is used to express the goal for each player. Thus, the way to check if one player has achieved the goal is to check whether the LTL formula is satisfied.

2.3.2 CTL* (Computation Tree Logic*)

CTL* stands for Computational Tree Logic*. It is considered to be a superset of linear temporal logic (LTL) and computational tree logic (CTL). In LTL, the semantics is defined as a set of executions, while in CTL the semantics is defined in terms of states.

Syntax CTL* combines temporal operators (X, F, G, U, and R) in LTL and path quantifiers (A and E) in CTL, thus is more powerful than any of the two. The two path quantifiers are defined as follows [15]:

- A = 'along all paths': universal path quantifier
- E = 'along at least (there exists) one path' : existential path quantifier

To give an example, consider the following CTL* formula: ' $E \phi_1 A \phi_2$ '. Here, ϕ_1 and ϕ_2 are LTL formulas. This formula states that there exists at least one path such that ϕ_1 has to be true, and ϕ_2 has to be true along all paths.

In this project, CTL* is used to perform synthesis checking.

2.3.3 SL (Strategy Logic)

In graph games, the type of game can be divided into the zero-sum game and non-zero-sum game. For the zero-sum game, the researchers have introduced logic to represent and formulate the games, such as temporal logic ATL [15]. Some extensions have also been proposed, such as ATL*, to compromise the limitation in ATL. However, these logics are not suitable for non-zero-sum games due to the limitation on the expressive power. Thus, strategy logic is introduced, which is designed to quantify over strategies in multi-agent concurrent systems [13].

Syntax Strategy Logic extends LTL(Linear-time Temporal Logic) with three additional operators:

- $\ll x \gg$ = existential strategy quantifier : indicates there exists a strategy x
- $\llbracket x \rrbracket$ = universal strategy quantifier : indicates for all strategies x
- (a,x) = agent binding : indicates agent a has a strategy associated with variable x

To give an example, consider the following SL formula :

$$\ll x \gg \ll y \gg \llbracket z \rrbracket (A,x)(B,y)(X \neg fail) \text{ and } (B,z)(G \text{ success})$$

Here, the formula states that there exists a strategy for agent A associated with variable x, a strategy for agent B associated with variable y such that the condition ‘false’ will not be met in the next state. Meanwhile, for all strategies associated with variable z for agent B, the condition ‘success’ will eventually be met sometimes in the future.

It is possible to express both zero-sum game and non-zero-sum game in strategy logic, including game logic (and thus module checking), Nash equilibrium, and secure equilibrium. In addition, it has been proved that strategy logic is decidable if all winning conditions are specified in linear temporal logic [13].

In this project, we use strategy logic to express the formulas that need to be checked in the algorithm.

2.4 Satisfiability

Within the application of automata theory in formal verification, two natural problems arise in the context of systems and their specifications: Satisfiability and Model Checking [16]. Satisfiability (SAT for short) is the problem of determining whether there exists any computational model that satisfies the given formula. The problem of satisfiability can be defined as follows:

Given formula ϕ , the question is: Is there a model π such that $\pi \models \phi$?

To give an example, consider the following Boolean formula f with variables x , y , and z :

$$f = (x \wedge y) \wedge (! z)$$

Intuitively, it is obviously to see that f returns true when $x = \text{true}$, $y = \text{true}$ and $z = \text{false}$. Thus, we say that f is satisfiable with the model $\{x : \text{true}, y : \text{true}, z : \text{false}\}$.

Satisfiability problem belongs to the decision problem, and is the first problem in the history that has been proved to be NP-complete. In average, the computational complexity is in polynomial time [17]. Several traditional methods have been used to solve the SAT problem, including enumeration method, local search method and the use of greedy algorithm. However, these methods are not suitable for solving large-scale SAT problems, due to the enormous search space.

SAT problem is a fundamental problem in Logic, as well as one of the core issues in Computer Science and Artificial Intelligence. Many problems can be transformed into SAT problem. So committed to find and develop a fast and effective algorithm to solve SAT problem has become an important topic in both theoretical research and practical implementations.

In this project, we aim to solve the satisfiability problem with formulas expressed in LTL. Several methods can be used to check LTL satisfiability. The most common one is to use reductions. This means that we can reduce the problem into one of the followings sub-problems: parity game, non-emptiness of APT automata, none-emptiness of other automata, and the brute-force approach.

2.5 Synthesis

Synthesis problem is also involved in this project, aiming to construct a system from a given specification. For instance, program synthesis is concerned with the following question: Given a specification, how can one develop an executable program so that the specification can be satisfied [18]?

For a given architecture A and an LTL specification ϕ , the synthesis problem for A and ϕ consists of finding an implementation S in A that realizes ϕ . To be more specific, the definition of synthesis problem can be expressed as follows in two-player game-theoretic terms:

Given formula ϕ , set of variables $\{v_1, v_2\}$, the question is:
Is there a strategy σ_1 for player1 such that for every strategy σ_2 for player2 we
have that $\pi(\sigma_1, \sigma_2) \models \phi$?

Here, $\{v_1, v_2\}$ represent the variables controlled by player1 and player2. σ_1, σ_2 indicate the available strategies that can be used by player1 and player2. The statement can be read as: Is there exist a winning strategy σ_1 for player1 against all strategies σ_2 for player2, such that the formula can be satisfied?

In synthesis problem, we can treat the two players as environment and system, where input variables are controlled by the environment, and output variables are controlled by the system. In our example, player1 is the environment, and player2 is the system. In this project, we aim to solve the synthesis problem with formulas expressed in LTL and CTL*.

Chapter 3

Algorithms

3.1 Overview

This chapter introduces the algorithm used in the implementation process. First, we define the unique problem that needs to be solved. Then, the game structure is presented. Last, the algorithm is introduced and explained in details.

3.2 Problem Definition

Nash Equilibria is one of the most fundamental concepts in game theory. In this project, we are interested in Nash-Equilibrium satisfiability, which is to verify whether a given game has a Nash equilibrium. In the paper “Imperfect Information in Reactive Modules Games”, checking the existence of Nash equilibria is also called as the NON-EMPTYNESS problem defined as follows [19]:

Given: G (Iterated Boolean Game with LTL goals)

NON-EMPTYNESS: Is it the case that $NE(G) \neq \emptyset$?

Although the NON-EMPTYNESS problem with LTL goals is undecidable for n -player games, it has been proved that the problem is decidable if we limit the number of players to two. With two players, the NON-EMPTYNESS problem can be further reduced to a series of temporal logic synthesis problem [19]. Thus, in this project, we focus on solving the synthesis problem for temporal logics, especially for LTL satisfiability, LTL synthesis, and CTL synthesis. The problem can be defined as follows:

Given Input : $\gamma_1(\text{LTL}), \gamma_2(\text{LTL}), AP_1, AP_2$

Question: Is there a Nash-Equilibrium for this game?

Here, $\gamma_1(\text{LTL}), \gamma_2(\text{LTL})$ represent the goals for player 1 and 2 that are expressed in LTL formulas, while AP_1, AP_2 represent the set of boolean variables controlled by each player.

3.3 The Game

In this section, we present the game structure used in the implementation and give an example of how the game is played.

Game Setting We use iterated boolean game with two players playing concurrently. For simplicity, both players receive perfect information, and only pure strategy is considered.

Game Structure The general structure of the game is presented as follows [4]:

$$G = (\{1, 2\}, \Phi, \Phi_1, \Phi_2, \gamma_1, \gamma_2)$$

Here, $\{1, 2\}$ represents player 1 and 2 respectively. Φ is a finite set of boolean variables which contains all variables that are controlled by either player. We can also write Φ as $\Phi_1 \cup \Phi_2$. Φ_1 is a finite set of boolean variables controlled by player1, and Φ_2 is a finite set of boolean variables controlled by player2. Finally, γ_1, γ_2 are the goals for player 1 and 2, formalized in LTL formulas.

To give an example, suppose we have two players, namely A and B. A controls 5 boolean variables $\{x_1, x_2, x_3, x_4, x_5\}$ and B controls 4 boolean variables $\{y_1, y_2, y_3, y_4\}$. The goal for A is $GF(x_1 \wedge x_5 \wedge \neg y_3)$, while the goal for B is $GF(\neg y_1 \wedge \neg y_2 \vee x_4)$. Based on the information given, we can describe the game as follows:

$$G = (\{1, 2\}, \Phi, \{x_1, x_2, x_3, x_4, x_5\}, \{y_1, y_2, y_3, y_4\}, GF(x_1 \wedge x_5 \wedge \neg y_3), GF(\neg y_1 \wedge \neg y_2 \vee x_4))$$

where $\Phi = \{x_1, x_2, x_3, x_4, x_5, y_1, y_2, y_3, y_4\}$.

How to Play To play the game, each player selects a subset of boolean variables that they control in each round. The selected variables will be assigned to true, while the remaining ones will be assigned to false. Figure 3.1 records the actions performed by the players in each round. Since the game is playing concurrently, one doesn't know his or her opponent's action in the current round. From the table, we can see that in Round 1, Player A chose $\{x_1, x_2\}$. This means that he has made the following assignment: $\{x_1, x_2 = true; x_3, x_4, x_5 = false\}$. Similarly, Player B chose $\{y_3\}$, indicating assignment $\{y_3 = true; y_1, y_2, y_4 = false\}$.



Player	Round 1	Round 2	Round 3	Round 4	Round 5	
A 	$\{x_1, x_2\}$	$\{x_1, x_3, x_5\}$	$\{x_2, x_5\}$	$\{x_1, x_4\}$	$\{x_2, x_3, x_4\}$
B 	$\{y_3\}$	$\{y_3, y_4, y_5\}$	$\{y_1, y_2\}$	$\{y_4\}$	$\{y_2, y_5\}$	

Figure 3.1: Actions performed by the players

After each player has finished their moves, the goal will be evaluated. Figure 3.2 shows how the variables are assigned based on the actions, and the evaluation results.

	Assignment										goal 1	goal2
	x1	x2	x3	x4	x5	y1	y2	y3	y4	y5		
Round 1	T	T	F	F	F	F	F	T	F	F	Satisfied	Unsatisfied
Round 2	T	F	T	F	T	F	F	T	T	T	Unsatisfied	Satisfied
Round 3	F	T	F	F	T	T	T	F	F	F	Unsatisfied	Unsatisfied
Round 4	T	F	F	T	F	F	F	F	T	F	Satisfied	Satisfied
Round 5	F	T	T	T	F	F	T	F	F	T	Satisfied	Unsatisfied
⋮												

Figure 3.2: Goal Evaluation

Since the goals are represented by LTL formulas, the result will be either true or false. True means that the goal is achieved, and false means that the goal is not achieved. In the next section, we are going to show how these evaluation results can be used in solving the ‘Existence of Nash Equilibria’ problem in our algorithm.

3.4 The Algorithm

The algorithm used in this project is adapted from the paper “Expressiveness and Complexity Results for Strategic Reasoning”. In the algorithm, the ‘Existence of Nash Equilibria’ problem is reduced to a series of temporal logic satisfiability and synthesis checking problems. The pseudo-code is presented as follows [5]:

Algorithm 1 Two-NE-Satisfiability($AP_1, AP_2, \gamma_1, \gamma_2$)

```

1: if SAT ( $\gamma_1 \wedge \gamma_2$ ) then return True
2: if SYN ( $\gamma_1, AP_1, AP_2$ ) or SYN ( $\gamma_2, AP_2, AP_1$ ) then return True
3: if SYN ( $\neg\gamma_2, AP_1, AP_2$ ) and SYN ( $\neg\gamma_1, AP_2, AP_1$ ) then return True
4: if SYN ( $E\gamma_1 \wedge A\neg\gamma_2, AP_1, AP_2$ ) or SYN ( $E\gamma_2 \wedge A\neg\gamma_1, AP_2, AP_1$ ) then return True
5: return False.

```

Syntax We write γ_1, γ_2 to represent the goals for Player 1 and 2. AP_1, AP_2 to represent the set of variables controlled by each player, SAT(γ) for checking the LTL satisfiability, and SYN(γ , In, Out) for checking the LTL and CTL* synthesis.

LTL Satisfiability *Line 1* checks for LTL satisfiability. Here, the goals for both players are combined into one LTL formula, $\gamma_1 \wedge \gamma_2$. In order to check whether the goal is achieved, we check whether the LTL formula is satisfied. In other words, we check whether the LTL formula returns true after evaluation. Obviously, if both players have their goals achieved, an equilibrium point is reached. This means that both players have reached their expectations, and are not willing to deviate from their original choices. Since this is a non-zero-sum game, we can say that a Nash equilibrium is reached when both players achieve their goals.

LTL Synthesis *Line 2 and 3* checks for LTL synthesis. *Line 2* states that if there exists a winning strategy for either player to achieve his or her own goal, then we have a Nash equilibrium. In the algorithm, it means that there exists a winning strategy for Player1 (or 2) such that after he assigned the values for the boolean variables he controls, no matter what moves Player2 (or 1) made, Player1's (or 2's) goal can always be achieved.

To give an explanation, consider the following two situations from Player1's point of view:

► **Situation 1:** Player1's goal is achieved and Player2's goal is not achieved. Since Player1 has a winning strategy, no matter what player2 does, he (Player2) can not prevent Player1 from achieving the goal. Thus, Player2 can't do any better to change the current situation. In the meantime, Player1 achieves the goal, so he reaches his expectation. Therefore, we say that a Nash equilibrium is reached when both players are satisfied with the current situation. In Logic, this situation can be formulated as:

$$\vec{\sigma} = \{(\sigma_1, \sigma_2) \mid \exists \sigma_1 (\forall \sigma_2)\}$$

read as, there exists some strategies for Player1, such that for all strategies that can be used by Player2, this strategy can be treated as the winning strategy for Player1. In other words, this means that by using this winning strategy, Player1 can both achieve his own goal as well as preventing Player2 from achieving the goal.

► **Situation 2:** Both players have their goals achieved. This is similar to check for LTL satisfiability. It means that although Player1 has a winning strategy, he can only make sure to achieve his own goal, but can't prevent Player2 to achieve the goal. Obviously, if both players have their goal achieved, both of them are happy with the result and won't regret their decisions. Thus, we say that a Nash equilibrium is reached. In Logic, this situation can be formulated as:

$$\vec{\sigma} = \{(\sigma_1, \sigma_2) \mid \exists \sigma_1 (\exists \sigma_2)\}$$

read as, there exists some strategies for Player1, such that for some strategies that can be used by Player2, this strategy can be treated as the winning strategy for Player1. In other words, it means that by using this winning strategy, Player1 can only ensure to achieve his own goal, but can't prevent Player2 from achieving the goal as well.

Figure 3.3 shows the strategy selection process regarding the two different situations. $\sigma_A^1 \dots \sigma_A^n$ is the set of strategies that can be used for Player1, and $\sigma_B^1 \dots \sigma_B^n$ is the set of strategies that can be used for Player2. It can be seen from the diagram that σ_A^i is a winning strategy for Player1 against all strategies for Player2 in the first situation. However, in the second situation, it is a winning strategy for Player1 against only one strategy for Player2, which is σ_B^i .

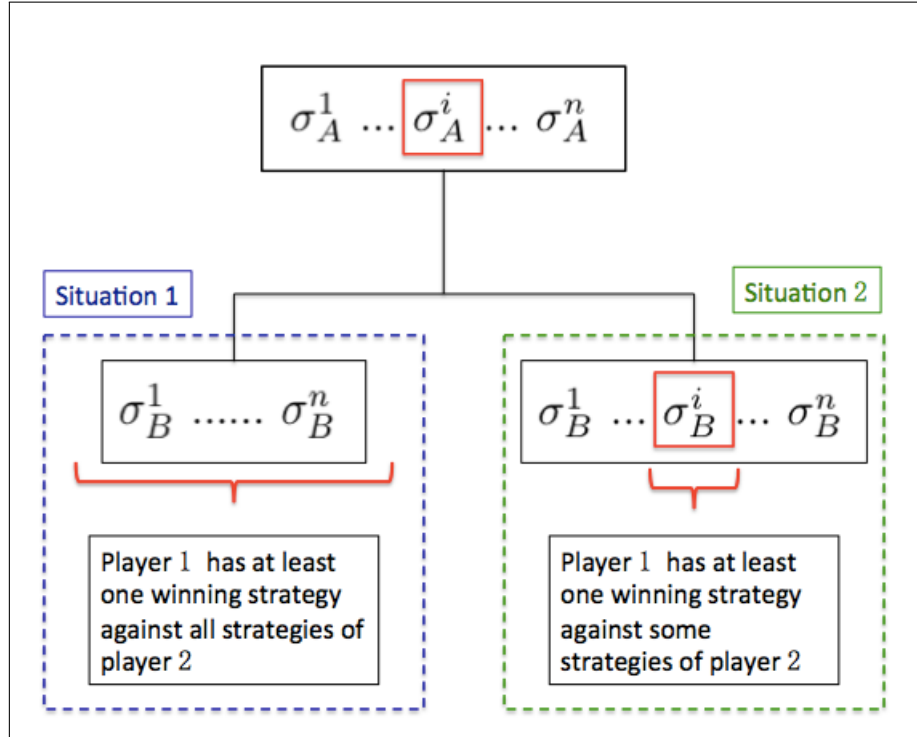


Figure 3.3: Strategy Selection Process

Notice that if one of the players has a winning strategy, the game already have a Nash equilibrium. Thus, the synthesis checking problems for Player 1 and 2 are independent. Here, Logic symbol ‘or’ is used to indicate the independence between the two.

Line 3 states that if there exists a winning strategy for both players to prevent the other from achieving the goal, then we have a Nash equilibrium. In this case, neither of the players will achieve their goal forever. Thus, no matter what they do, the current situation can not be changed. This is similar as having a deadlock in concurrent programming, where each thread is preventing the other from acquiring the lock.

Figure 3.4 presents the working principle of winning strategies for Player 1 and 2. $\sigma_A^1 \dots \sigma_A^n$ is the set of strategies used by Player1, and $\sigma_B^1 \dots \sigma_B^n$ is the set of strategies used by Player2. In the diagram, σ_A^i is the winning strategy for Player1, and σ_B^i is the winning strategy for Player2.

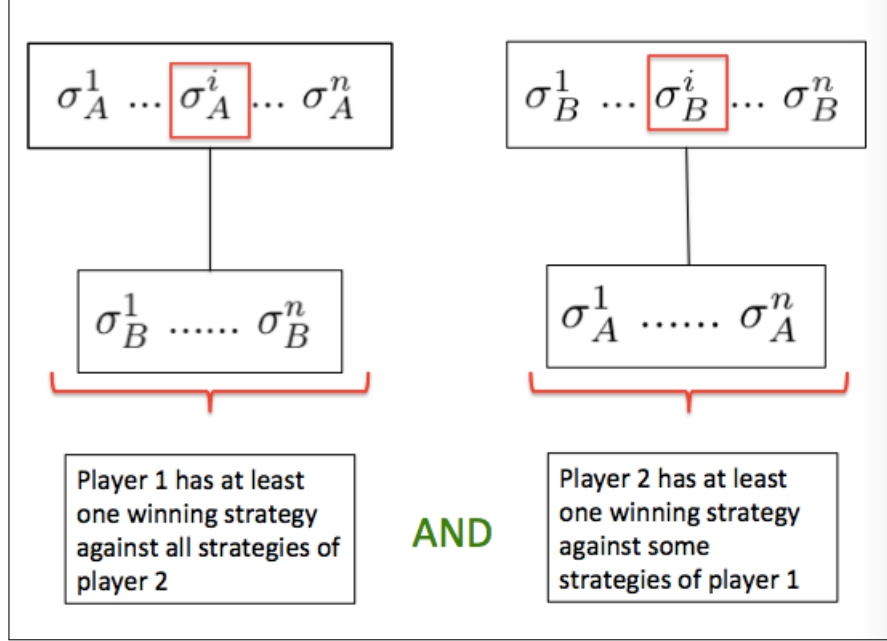


Figure 3.4: Working Principle of Winning Strategies for Player 1 and 2

In this case, we say the game has a Nash equilibrium only if both players have a winning strategy. Thus, the synthesis checking problems for Player 1 and 2 are related to each other. Here, Logic symbol ‘and’ is used to indicate the dependency between the two.

CTL* Synthesis *Line 4* checks for CTL* synthesis. If the result returns true, it means that Player 1 (or 2) has a winning strategy to prevent the other player from achieving the goal. Simultaneously, there exists some strategies for Player 1 (or 2) to achieve his own goal. The CTL* formula can be split into two segments. ‘ $E\gamma_1$ ’ indicates that there exists some winning strategies for Player1 to achieve the goal, and ‘ $A\neg\gamma_2$ ’ indicates that there exists some winning strategies for Player1 to prevent Player2 from achieving the goal against all strategies that can be used by Player2. Again, we can say that the game has a Nash equilibrium if the CTL* synthesis returns true for either player. Thus, the logic symbol ‘or’ is used here to indicate independent checking.

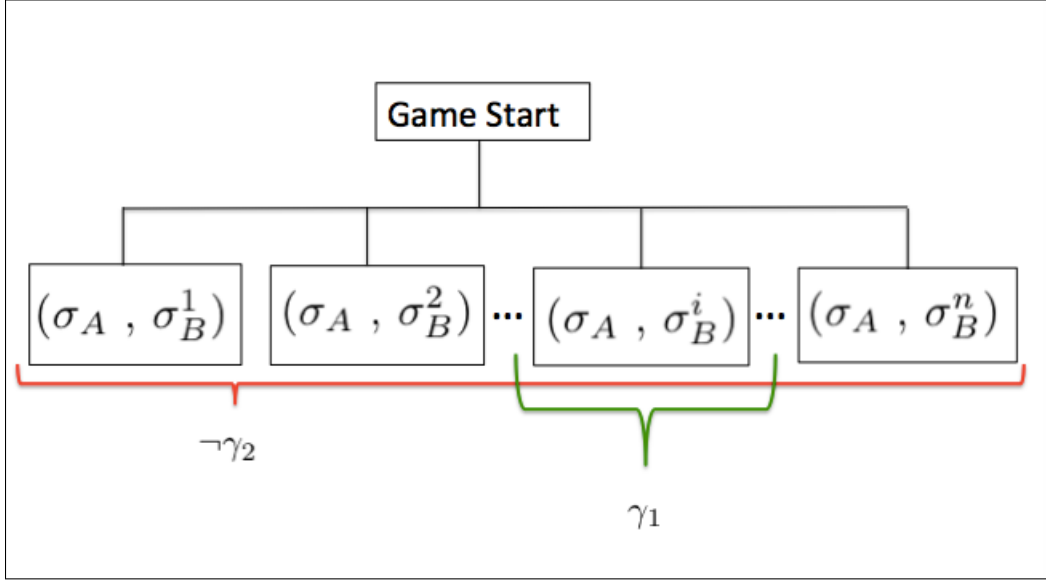


Figure 3.5: Working Principle of Winning Strategy σ_A

Figure 3.5 shows an example of how the winning strategy works. Here, σ_A is a winning strategy for Player1, and $\sigma_B^1 \dots \sigma_B^n$ is the set of strategies available for Player2. From the diagram, we can see that no matter what strategy ($\sigma_B^1 \dots \sigma_B^n$) Player2 chooses, σ_A can be used as the winning strategy of Player1 for the negation of Player2's goal ($\neg\gamma_2$). In addition, if Player2 uses strategy σ_B^i among all the strategies, σ_A can also be used as the winning strategy of Player1 to achieve his own goal (γ_1).

Chapter 4

Design Implementation

4.1 Overview

MCMAS tool is used in the implementation to perform LTL satisfiability, LTL synthesis and CTL* synthesis checking. More specifically, we used MCMAS-SL in this project, which is an extension of MCMAS.

Python is chosen as the programming language, due to its high coding flexibility and well-developed libraries. The program is designed in a user-friendly way, meaning that it is not hard-coded, and has the flexibility to adapt different types of user inputs. The implementation details will be explained in section 4.3.

In this chapter, we introduce the model checking tool: MCMAS and its extension MCMAS-SL. We will also describe the detailed implementation of the algorithm with the use of MCMAS, and discuss two design extensions.

4.2 Introduction to MCMAS

4.2.1 What is MCMAS ?

MCMAS is an open source model checking toolkit for Multi-Agent Systems(MAS). It permits the automatic verification of specifications that use the standard temporal modalities [20]. The tool is written in C/C++, and is platform dependent [20]. Figure 4.1 shows the implementation structure of MCMAS [20]. The tool takes input written in ISPL specification and a set of formulas that need to be verified. Then it evaluates the given formulas using algorithms based on OBDDs and return a boolean result (true or false) [21].

Interpreted systems Interpreted Systems provides the formal semantics for MCMAS program [20]. The tool uses ISPL (Interpreted Systems Programming Language) as the modelling language to formalize the system specification. Each agent is characterized by a set of local states and actions as well as a local protocol. An evolution function is used to determine how the local state varies with different local actions [20].

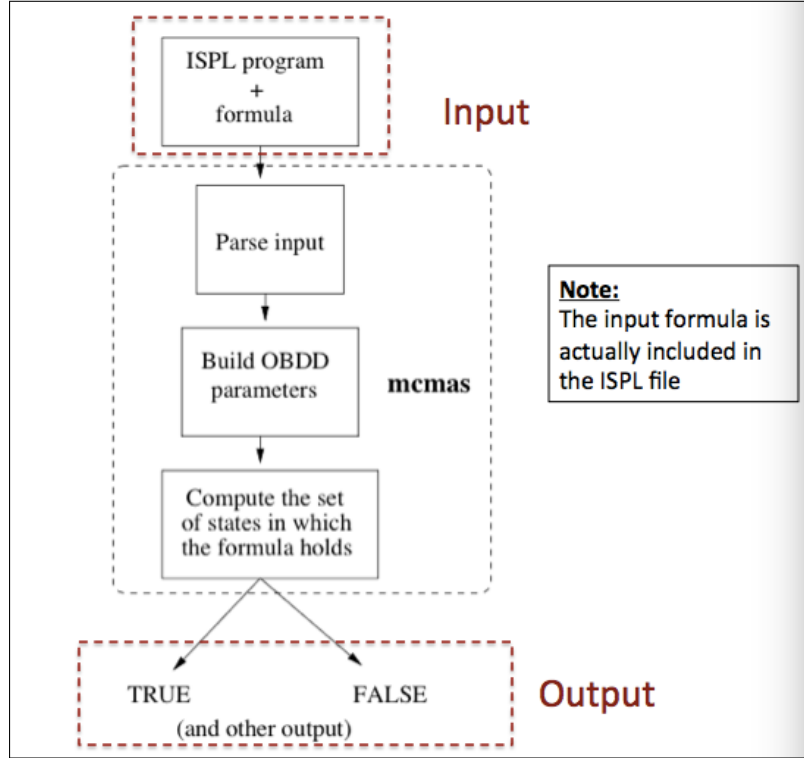


Figure 4.1: Implementation Structure of MCMAS

Agent In MCMAS, there are two kinds of agent, standard agent and environment agent. Each player can be considered as a standard agent. The environment agent is optional in ISPL file, which aims to describe the overall boundary conditions in the system. Figure 4.2 is an example of an Agent in ISPL specification. Each agent is characterized by 4 terms: local states, actions, protocol, and the evolution function [21].

Local states Each agent has its own local variables. Each local variable can be assigned with different values. Thus, any possible combination of assignments can be referred as a local state. For instance, suppose one agent has two boolean local variables, namely A and B. In this case, there are 4 possible local states:

$$\{A : True, B : True\}, \{A : True, B : False\}, \{A : False, B : True\}, \{A : False, B : False\}$$

```

22  Agent Player
23    Vars:
24      x : boolean;
25    end Vars
26    Actions = {ac0, ac1};
27    Protocol:
28      x = false: {ac0};
29      x = true: {ac1};
30    end Protocol
31    Evolution:
32      (x = false) if Action=ac0;
33      (x = true) if Action=ac1;
34    end Evolution
35  end Agent

```

Figure 4.2: Sample Agent defined in ISPL code

Local states are private and can only be observed by their own agent. Each agent can't see others' local variables, unless the variables are defined as local observation variables in environment agent. Local observation variables can be modified only by the environment, and are read-only for all other agents [21].

Actions The specific actions performed by the agent are defined under this section. These actions will be used in the Evolution function, and are visible to all other agents.

Protocol The purpose of protocol is to give instructions to the agent on which action to perform, based on the assignment of local variables. We can treat the protocol like a FSM (Finite State Machine). Based on different conditions (assignments of local variables), we will enter different states and perform different actions. Notice that the protocol is non-deterministic, which means that actions can be randomly performed.

Evolution function The evolution function assigns new values to local variables regarding to the specific action that the agent performs. Since actions are chosen based the previous assignment of local variables, the evolution function can also be considered as an update function on local variables.

Notice that MCMAS adapts memoryless version. This means the local states of each agent don't include the local history of the previous running results. As a result, strategies are also considered to be memoryless only. This is mainly because of the high computational

complexity of the problems that the tool needs to solve [22].

MCMAS can be run either from the command line or from the GUI(Graphical User Interface) [21]. In this project, we run MCMAS by giving instruction in the command line, which allows more freedom on parsing the output, and customizing the running options.

4.2.2 Model Checker: MCMAS-SL

There are several extension versions of MCMAS, namely MCMAS-1G, MCMAS-SL, and MCMAS-SLK. In this project, we use MCMAS-SL.

The working principle for MCMAS and MCMAS-SL is almost the same, except for the formula specification. MCMAS adapts CTL (Computational Tree Logic) formulas, while in MCMAS-SL the formulas are expressed in SL (Strategy Logic).

In system design and verification, researchers are interested in modelling logics for strategic behaviour in multi-agent systems. Two important logics have been brought up, one is ATL^* , and the other is SL_{CHP} [15]. Both of them are very expressive and have the powerful framework for reasoning explicitly about strategies [15]. However, ATL^* suffers from a number of limitations for reasoning in multi-agent systems [22], and SL_{CHP} can't fully capture the system specification in the multi-agent setting [15]. For instance, it is not possible to bind strategies to various agents, or to bind the same agent with different strategies [22]. Thus, Strategy Logic is introduced. It is a proper extension of both ATL^* and SL_{CHP} , which has been proved to be powerful enough to solve the Nash equilibrium problem.

In MCMAS-SL, the formulas that need to be checked are expressed in a variant of SL (Strategy Logic) [22]. Similar to MCMAS, MCMAS-SL only considers the memoryless system. One main advantage about MCMAS-SL is its ability to solve strategy synthesis problem, which defines as:

‘Given an interpreted system, an initial state, and a formula expressed in SL, the problem is to find a model that can satisfy the formula’ [22]

In our algorithm, we converted the Nash equilibrium problem to temporal logic synthesis problem. Thus, MCMAS-SL is a perfect tool to use in the implementation process. For the rest of the report, we will call MCMAS-SL as MCMAS for consistency.

4.2.3 ISPL File

The input file required by MCMAS is written in ISPL specification. Figure 4.3 is an example of the ISPL input file [21]:

```
Evaluation
  recbit if ( (Receiver.state=r0) or (Receiver.state=r1) );
  recack if ( ( Sender.ack = true ) );
  bit0 if ( (Sender.bit=b0));
  bit1 if ( (Sender.bit=b1) );
  envworks if ( Environment.state=SR );
end Evaluation

InitStates
  ( (Sender.bit=b0) or (Sender.bit=b1) ) and
  ( Receiver.state=empty ) and ( Sender.ack=false) and
  ( Environment.state=none );
end InitStates

Groups
  g1 = {Sender,Receiver};
end Groups

Fairness
  envworks;
end Fairness

Formulae
  AF(K(Sender,K(Receiver,bit0) or K(Receiver,bit1)));
  AG(recack -> K(Sender,(K(Receiver,bit0) or K(Receiver,bit1))));
end Formulae
```

Figure 4.3: Sample ISPL file

Despite of agents, the ISPL file also includes the following five sections: Evaluation, InitStates, Groups, Fairness and Formulae [21]. In the evaluation section, boolean variables are introduced and assigned with values true or false, based on the local state of the current agent. The InitStates section declares all the possible initial states for the local state. The Groups section combines several agents together in the same group. The Fairness section contains a list of formulas that must return true in the whole execution process. Finally, the Formulae section contains a list of SL (Strategy Logic) formulas that need to be verified by the tool.

4.2.4 MCMAS Simulation

Once the input file is ready, we can run MCMAS from the command line with the following command:

```
$/mcmas input.ispl
```

This minimal MCMAS execution consists of the invocation of the executable followed by the name of the input ISPL file [21]. Figure 4.4 shows an example of MCMAS output [21]:

```
$ ./mcmas examples/bit_transmission_protocol.ispl
*****
MCMAS v1.2.2

This software comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

Please check http://vas.doc.ic.ac.uk/software/mcmas/ for the latest release.
Report bugs to <mcmas@imperial.ac.uk>
*****

examples/bit_transmission_protocol.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building OBDD for initial states...
Building reachable state space...
Checking formulae...
Building set of fair states...
Verifying properties...
  Formula number 1: (AF K(Sender, (K(Receiver, bit0) || K(Receiver, bit1)))), is TRUE in the model
  Formula number 2: (AG (recack -> K(Sender, (K(Receiver, bit0) || K(Receiver, bit1)))), is TRUE in the model
done, 2 formulae successfully read and checked
execution time = 0
number of reachable states = 18
```


Output 

Figure 4.4: MCMAS Output

MCMAS will first perform a detailed syntax checking on the input ISPL file. If an error pops up, the error message will be returned in the terminal window, and the program terminates. Otherwise, if the syntax checking is successful, MCMAS will perform the formal verification on the set of formulas defines in the ISPL file. For each independent formula, MCMAS will return a boolean output, indicating whether the formula can be satisfied or not with the given model.

4.3 Algorithm Implementation

As mentioned in Chapter 3, the algorithm used converts Nash equilibrium problem to a set of temporal logic synthesis problems, more specifically, to LTL Satisfiability and LTL/CTL* synthesis problems. In the implementation, we use MCMAS model checker to solve the synthesis problem.

Figure 4.5 presents the system-level design flow. The implementation process can be divided into 6 steps. First, the program reads in the user input file and use input parser to extract useful information. If the input file has a wrong format, an error message will be returned to the user. Then, the program uses the given information to construct an iterated Boolean game, and generates an ISPL file for this game. With the ISPL input file, the program runs MCMAS model checker and collects the produced output. Finally, the program analyzes the output and returns the final result to the user.

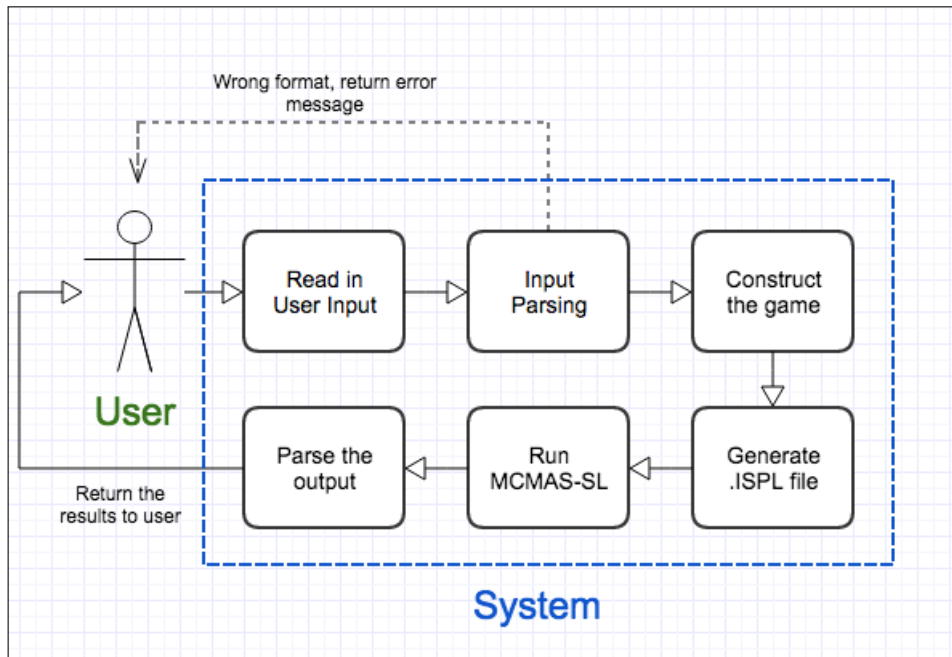


Figure 4.5: System-level Design

The main implementation contains three python files:

- `main.py`: This file defines the main program
- `sourcecode.py`: This file contains all the source code implementation
- `test.py`: This file includes all the test cases.

All the files can be retrieved from Appendix A.

4.3.1 Read User Input File

The first step is to read in the user input from file ‘input.ibg’. Here, ‘ibg’ is short for iterated Boolean game. In order to construct a valid game, the user needs to provide the following information:

1. Set of boolean variables controlled by each player (each separated by semicolon)
2. Goals for each player (expressed in LTL formula)
3. Explore option (ALL/WHICH)

```
1  ### User Input File ###
2
3  Player1: x;y
4  Player2: z
5  Goal1: G F (x and y)
6  Goal2: X (!z)
7  Explore: ALL
8
9
10
```

Variables

Goals

Default extension setting

Figure 4.6: User Input File

Figure 4.6 shows a sample user input file. Based on the information given, we can construct the following game: $G = (\{1, 2\}, \{x, y, z\}, \{x, y\}, \{z\}, G F (x \text{ and } y), X(!z))$

In python code, the input parser works as follows: First, the file content is stored as a complete string. Then, the string is split by different keywords(‘Player 1’, ‘Player 2’, ‘Goal 1’, ‘Goal 2’ and ‘Explore’) that are pre-defined in the user input file. It is required by the parser that each variables are separated by a semicolon. Finally, the parser stores the set of variables as lists, and goals as strings. For detailed implementation, please refer to Appendix A.1.

4.3.2 Translate to ISPL File

After reading in the input information, the next step is to generate the ISPL file which is required by MCMAS. This is the core of our implementation.

4.3.2.1 Agents

In a two-player game, there are three agents: Environment, Player1, and Player2. Since Player1 and Player2 share the same resources, we combined them into one category below named ‘Players’ for simplicity.

Environment In concurrent-game settings, there is no need to have an environment. Since it is required by ISPL, we define the environment as a dummy agent. Figure 4.7 shows the (dummy) Agent Environment coded in ISPL:

```
8  Agent Environment
9  Obsvars:
10   dummy : boolean;
11  end Obsvars
12  Actions = {none};
13  Protocol:
14   Other : {none};
15  end Protocol
16  Evolution:
17   dummy = true if Action=none;
18  end Evolution
19  end Agent
```

Figure 4.7: Agent Environment

Here, the agent has one dummy boolean variable, and performs no actions. Notice that the dummy boolean variable will not be used in the formulas. In python code, the Agent Environment is generated by function ‘createEnvironment’. Detailed implementation can be referred to Appendix A.

Players In our game, there are two players, namely Player1 and Player2. Figure 4.8 shows an example of Agent Player1 coded in ISPL. The player has one local variable (x), and two actions (ac0, ac1). First, one action will be randomly selected in the Protocol section. Then, the local variable x will be randomly assigned in the Evolution section, based on the selected action.

```

22 Agent Player1
23   Vars:
24     x : boolean;
25   end Vars
26   Actions = {ac0, ac1};
27   Protocol:
28     Other : {ac0, ac1};
29   end Protocol
30   Evolution:
31     (x = false) if Action=ac0;
32     (x = true) if Action=ac1;
33   end Evolution
34 end Agent

```

Figure 4.8: Agent Player with one variable

```

22 Agent Player1
23   Vars:
24     x : boolean;
25     y : boolean;
26   end Vars
27   Actions = {ac00, ac01, ac10, ac11};
28   Protocol:
29     Other : {ac00, ac01, ac10, ac11};
30   end Protocol
31   Evolution:
32     (x = false) and (y = false) if Action=ac00;
33     (x = false) and (y = true) if Action=ac01;
34     (x = true) and (y = false) if Action=ac10;
35     (x = true) and (y = true) if Action=ac11;
36   end Evolution
37 end Agent

```

Figure 4.9: Agent Player with two variables

```

22 Agent Player1
23   Vars:
24     x : boolean;
25     y : boolean;
26     z : boolean;
27   end Vars
28   Actions = {ac000, ac001, ac010, ac011, ac100, ac101, ac110, ac111};
29   Protocol:
30     Other : {ac000, ac001, ac010, ac011, ac100, ac101, ac110, ac111};
31   end Protocol
32   Evolution:
33     (x = false) and (y = false) and (z = false) if Action=ac000;
34     (x = false) and (y = false) and (z = true) if Action=ac001;
35     (x = false) and (y = true) and (z = false) if Action=ac010;
36     (x = false) and (y = true) and (z = true) if Action=ac011;
37     (x = true) and (y = false) and (z = false) if Action=ac100;
38     (x = true) and (y = false) and (z = true) if Action=ac101;
39     (x = true) and (y = true) and (z = false) if Action=ac110;
40     (x = true) and (y = true) and (z = true) if Action=ac111;
41   end Evolution
42 end Agent

```

Figure 4.10: Agent Player with three variables

Since the values of local variables are assigned randomly, all possible combinations of assignments need to be considered in the Evolution section. To achieve this, we pair every possible assignment with a unique action. The relationship can be summarized as:

$$n \text{ variables} \implies 2^n \text{ actions.}$$

meaning if the agent has n variables, there are in total 2^n actions. In the Protocol section, the keyword ‘Other’ is used to indicate that one action will be randomly selected from the action set despite of the previous assignment of variables. Figure 4.8, 4.9, and 4.10 present the Agent Player1 coded in ISPL with one, two and three local variables respectively. We can see that one variable requires two(2^1) actions, two variables requires four(2^2) actions, and three variables requires eight(2^3) actions.

```

30 def createPlayer(fo,var,num):
31     fo.write("Agent Player1\n") if num == "1" else fo.write("Agent Player2\n")
32
33     # action list
34     seqList = ["".join(seq) for seq in itertools.product("01", repeat = len(var))]
35     actionList = ["ac" + seq for seq in seqList] # store actions
36
37     # variables
38     fo.write("  Vars:\n")
39     for i in var:
40         fo.write("    " + i + " : boolean;\n")
41     fo.write("  end Vars\n")
42
43     # actions
44     fo.write("  Actions = {" + ", ".join(actionList) + "};\n")
45
46     # protocols
47     fo.write("  Protocol:\n")
48     fo.write("    Other : {" + ", ".join(actionList) + "};\n")
49     fo.write("  end Protocol\n")
50
51     # evolution
52     fo.write("  Evolution:\n")
53     for i in range(len(seqList)):
54         fo.write("    ")
55         for j in range(len(var)):
56             fo.write("(" + var[j] + " = " + str(bool(int(seqList[i][j])))<
57             .lower() + ")")
58             if (j == len(var) - 1): break
59             fo.write(" and ")
60         fo.write(" if Action=" + actionList[i] + ";\n")
61     fo.write("  end Evolution\n")
62     fo.write("end Agent\n")

```

Figure 4.11: Function *createPlayer()*

In python code, the Agent Player is generated by function *createPlayer()*. Figure 4.11 is a screenshot of the source code. Python module *itertools* is used to achieve efficient looping. We use *itertools.product(*iterables, repeat=n)* function to generate the binary sequence list (@Line 31). Here, the input iterables are binary bits ‘01’, and n is the number of variables. To give an example, if there are three variables, the sequence list should contain all combinations of 3-bit binary numbers: {000,001,010,...,111}. Then, the sequence list is converted to action set by adding the prefix ‘ac’ to each element in the list (@Line 32). The binary number contained in each action is used to indicate the unique assignment of local variables (@Line 53), with ‘0’ indicating false and ‘1’ indicating true. For instance, ‘ac10’ is paired with assignment ‘x = true, y = false’.

4.3.2.2 Evaluation

In this section, we define the boolean variables that will be used in the Formulae section. Since all the local variables are private to other agents, we redefine them with the same variable name. Consider the example ISPL code in Figure 4.12 :

```

68  Evaluation
69    x if Player1.x = true;
70    y if Player1.y = true;
71    z if Player1.z = true;
72    u if Player2.u = true;
73    v if Player2.v = true;
74    w if Player2.w = true;
75  end Evaluation

```

Figure 4.12: Evaluation coded in ISPL

Here, Player1 controls variables x, y, and z, while Player2 controls variables u, v, and w. Notice that ‘x’ and ‘Player1.x’ are two different variables. ‘x’ is treated as a global variable that can be observed by anyone, while ‘Player1.x’ is treated as a private variable that can be observed only by Player1. In python code, the Evaluation section is generated by function *createEvaluation()*. Detailed implementation can be referred to Appendix A.1.

4.3.2.3 Initstates

In this section, we declare the initial state of local states for each agent. Based on the algorithm, the initial values should be assigned randomly. To achieve this, we include all possible combinations of states in the Initstates section. Figure 4.13 shows an example of Initstates coded in ISPL:

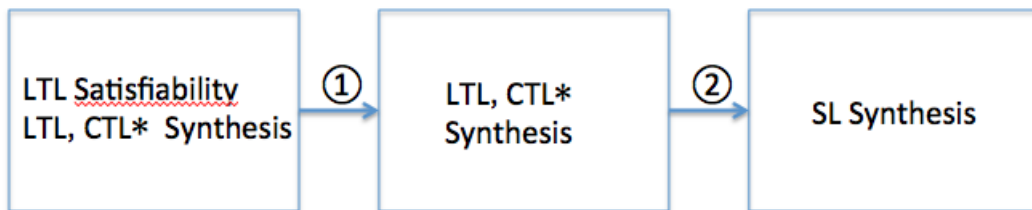
```
62 InitStates
63   (Environment.dummy = true) and
64   ((Player1.x = true) or (Player1.x = false)) and
65   ((Player1.y = true) or (Player1.y = false)) and
66   ((Player2.z = true) or (Player2.z = false));
67 end InitStates
```

Figure 4.13: Initstates coded in ISPL

Here, Player1 controls variables ‘x, y’ and Player2 controls variable ‘z’. Thus, there are eight possible initial states. In each round, MCMAS will randomly pick one of the initial states as the new starting point. In python code, the Initstates section is generated by function *createInitstates()*. Detailed implementation can be referred to Appendix A.1.

4.3.2.4 Formulae

In this section, we write the formulas that need to be verified. For MCMAS-SL model checker, the formula is formalized in strategy logic. Thus, before generating the ISPL code, we need to convert our algorithm (presented in Chapter 3) into strategy logic synthesis problems. The process can be divided into two steps described as follows:



In the first step, LTL satisfiability problem is converted to LTL synthesis problem.

The updated pseudo-code is presented as follows:

Algorithm 2 Two-NE-Satisfiability($AP_1, AP_2, \gamma_1, \gamma_2$)

Require: SAT translating to SYN

- 1: **if** SYN $(\forall(\gamma_1 \wedge \gamma_2), \emptyset, AP_1 \cup AP_2)$ **then return** *True*
 - 2: **if** SYN $(\forall\gamma_1, AP_1, AP_2)$ or SYN $(\forall\gamma_2, AP_2, AP_1)$ **then return** *True*
 - 3: **if** SYN $(\forall\neg\gamma_2, AP_1, AP_2)$ and SYN $(\forall\neg\gamma_1, AP_2, AP_1)$ **then return** *True*
 - 4: **if** SYN $(E\gamma_1 \wedge A\neg\gamma_1, AP_1, AP_2)$ or SYN $(E\gamma_2 \wedge A\neg\gamma_2, AP_2, AP_1)$ **then return** *True*
 - 5: **return** *False*.
-

In the original algorithm, *Line 1* represents LTL satisfiability SAT $(\gamma_1 \wedge \gamma_2)$, meaning that the formula returns true if both goals are satisfied. Here, we convert this into a synthesis problem, by having no input variable, and output variables as the union of variables controlled by each player.

In the second step, LTL and CTL* synthesis problems are converted to strategy logic synthesis problems. The updated pseudo-code is presented as follows:

Algorithm 3 Two-NE-Satisfiability($AP_1, AP_2, \gamma_1, \gamma_2$)

Require: Translating to Strategy Logic synthesis problems

- 1: **if** $\ll strategy_p1 \gg (Player1, strategy_p1) \ll strategy_p2 \gg (Player2, strategy_p2) (\gamma_1 \wedge \gamma_2)$ **then return** *True*
 - 2: **if** $\ll strategy_p1 \gg (Player1, strategy_p1) \ll strategy_p2 \gg (Player2, strategy_p2) (\gamma_1)$ **then return** *True*
 - 3: **if** $\ll strategy_p2 \gg (Player2, strategy_p2) \ll strategy_p1 \gg (Player1, strategy_p1) (\gamma_2)$ **then return** *True*
 - 4: **if** $\ll strategy_p1 \gg (Player1, strategy_p1) \ll strategy_p2 \gg (Player2, strategy_p2) (!\gamma_2)$ and $\ll strategy_p2 \gg (Player2, strategy_p2) \ll strategy_p1 \gg (Player1, strategy_p1) (!\gamma_1)$ **then return** *True*
 - 5: **if** $\ll strategy_p1 \gg (Player1, strategy_p1) \ll strategy_p2 \gg (Player2, strategy_p2) !\gamma_2$ and $\ll strategy_p2 \gg (Player2, strategy_p2) \gamma_1$ **then return** *True*
 - 6: **if** $\ll strategy_p2 \gg (Player2, strategy_p2) \ll strategy_p1 \gg (Player1, strategy_p1) !\gamma_1$ and $\ll strategy_p1 \gg (Player1, strategy_p1) \gamma_2$ **then return** *True*
 - 7: **return** *False*.
-

Here, $\langle\langle\rangle\rangle$ indicates ‘there exists such strategy’, and $\llbracket\rrbracket$ indicates ‘for all possible strategies’. *Line 1* checks for LTL satisfiability, states that ‘there exist a winning strategy for both players to achieve their goals.’ *Line 2,3,4* check for LTL synthesis. *Line 2* states that ‘there exist a winning strategy for Player1 to achieve his goal against all possible strategies available for Player2’. *Line 3* states that ‘there exist a winning strategy for Player2 to achieve his goal against all possible strategies for available Player1’. *Line 4* states that ‘there exist a winning strategy for Player1 to the negation of Player2’s goal against all possible strategies available for Player2, and vice versa’. The last two lines check for CTL* synthesis. *Line 5* states that ‘there exist a winning strategy for Player1 against all possible strategies for Player2 to the negation of Player2’s goal , and against some strategies for Player2 to achieve his own goal’. *Line 6* states that ‘there exist a winning strategy for Player2 against all possible strategies for Player1 to the negation of Player1’s goal , and against some strategies for Player1 to achieve his own goal’.

As mentioned in Chapter 3, the symbol ‘or’ and ‘and’ in the pseudo-code are not logic symbols. Thus, when translating to ISPL, we can’t group all formulas together. Indeed, each formula needs to be checked independently. Figure 4.14 shows the Formulae section coded in ISPL:

```

65  Formulae
66  -- LTL SAT
67  <<strategy_env>> (Environment, strategy_env) (
68    ( <<strategy_p1>> (Player1, strategy_p1) <<strategy_p2>> (Player2, strategy_p2)
69      X ( (G F (x and y) ) and (G F (!x and !y)) ) )
70  );
71  -- LTL SYN
72  <<strategy_env>> (Environment, strategy_env) (
73    ( <<strategy_p1>> (Player1, strategy_p1)  $\llbracket$ strategy_p2 $\rrbracket$  (Player2, strategy_p2) X (G F (x and y) ) )
74  );
75  <<strategy_env>> (Environment, strategy_env) (
76    ( <<strategy_p2>> (Player2, strategy_p2)  $\llbracket$ strategy_p1 $\rrbracket$  (Player1, strategy_p1) X (G F (!x and !y)) )
77  );
78  -- LTL SYN
79  <<strategy_env>> (Environment, strategy_env) (
80    ( <<strategy_p1>> (Player1, strategy_p1)  $\llbracket$ strategy_p2 $\rrbracket$  (Player2, strategy_p2) X (!(G F (!x and !y)) ) )
81  );
82  <<strategy_env>> (Environment, strategy_env) (
83    ( <<strategy_p2>> (Player2, strategy_p2)  $\llbracket$ strategy_p1 $\rrbracket$  (Player1, strategy_p1) X (!(G F (x and y) ) ) )
84  );
85  -- CTL* SYN
86  <<strategy_env>> (Environment, strategy_env) (
87    ( <<strategy_p1>> (Player1, strategy_p1) (  $\llbracket$ strategy_p2 $\rrbracket$  (Player2, strategy_p2) X (!(G F (!x and !y)) ) and
88      <<strategy_p2>> (Player2, strategy_p2) X (G F (x and y) ) ) )
89  );
90  <<strategy_env>> (Environment, strategy_env) (
91    ( <<strategy_p2>> (Player2, strategy_p2) (  $\llbracket$ strategy_p1 $\rrbracket$  (Player1, strategy_p1) X (!(G F (x and y) ) ) and
92      <<strategy_p1>> (Player1, strategy_p1) X (G F (!x and !y)) ) )
93  );
94  end Formulae

```

Figure 4.14: Formulae coded in ISPL

Here, we can see that the algorithm is divided into seven independent formulas, each wrapped by a dummy environment. In python code, the Formulae section is generated by function *createFormulae()*. Detailed implementation can be referred to Appendix A.1.

4.3.3 Run MCMAS

After generating the ISPL input file, we are ready to run the MCMAS model checker.

Figure 4.15 shows a screenshot of python source code for running MCMAS:

```

220 def runMCMAS(filename):
221     start_time = time.time()
222     print("%s:running MCMAS..." % start_time)
223     print("-----")
224
225     # Call mcmas
226     os.chdir("../..") # change directory
227     output = os.popen("./mcmas examples/iBG/" + filename).read()
228
229     # Set flag
230     done = False
231
232     # track excution time
233     print output
234     print("-----")
235     print("%s:finishing MCMAS..." % time.time())
236     print("-----")
237     end_time = time.time() - start_time
238     print("Execution Time:%s seconds " % end_time)
239     print("-----")
240
241     f = open('examples/iBG/excutiontime.ibg','a+')
242     f.write("Execution Time:%s seconds\n" % end_time)
243     f.close()

```

Figure 4.15: Function *runMCMAS()*

Here, the output produced by MCMAS is traced and stored for later use in result parsing (@Line227). Time stamps are used to track the execution time for performance evaluation. Since MCMAS also produce its own execution time as one of the outputs, we build up some test cases to compare the two. Based on the experimental results, it can be concluded that when the system is small (for simple and easy game), the execution time tracked by time stamps dominates in runtime. When the system is large (for complex and time-consuming game), the execution time produced by MCMAS dominates in runtime and the program running time can be negligible.

4.3.4 Parse the Output

MCMAS distinguishes each independent formula by assigning a unique number. For instance, the first formula in the list will be titled as ‘Formula number 1’, the second formula will be titled as ‘Formula number 2’ and so on. In total, there are seven formulas. Formula number 1 checks for LTL Satisfiability. Formula number 2-5 check for LTL Synthesis. Finally, formula number 6-7 check for CTL* Synthesis. Figure 4.16 shows a screenshot of the source code:

```
245     #identity filename
246     if filename == "input.ispl":
247         # Parse output
248         result1 = re.search("Formula number 1(.*?)model", output)
249         result2 = re.search("Formula number 2(.*?)model", output)
250         result3 = re.search("Formula number 3(.*?)model", output)
251         result4 = re.search("Formula number 4(.*?)model", output)
252         result5 = re.search("Formula number 5(.*?)model", output)
253         result6 = re.search("Formula number 6(.*?)model", output)
254         result7 = re.search("Formula number 7(.*?)model", output)
255
256         if (("is TRUE" in result1.group(1)) or ("is TRUE" in result2.group(1)) or
257             ("is TRUE" in result3.group(1)) or ("is TRUE" in result6.group(1)) or
258             ("is TRUE" in result7.group(1))):
259             # raise flag
260             done = True
261             # print result
262             print "The game has a Nash Equilibrium "
263         elif ("is TRUE" in result4.group(1)) and ("is TRUE" in result5.group(1)):
264             # raise flag
265             done = True
266             # print result
267             print "The game has a Nash Equilibrium "
268         else:
269             print "The game doesn't have a Nash Equilibrium "
```

Figure 4.16: Output Parser

Remember that in the original algorithm, formula number 4 and 5 are connected with symbol ‘and’, while others are connected with symbol ‘or’. Thus, we say the game has a Nash equilibrium if:

1. Either one of the ‘Formula number 1, 2, 3, 6, or 7’ returns True. **or**
2. Both ‘Formula number 5 and 6’ return True.

In python code, this is achieved by using regular expression operations (Python 3). *re.search()* is used to parse the output for each formula (@Line 248-254), while *groups()* is used to extract the results (@Line 256, 263).

4.4 Design Extensions

4.4.1 Explore: ALL vs WHICH

The ‘Explore’ feather is developed to allow the user to choose different ways in exploring and checking formulas while running MCMAS. There are two options: ALL and WHICH. The user is allowed to select either option in the user input file (input.ibg).

ALL In this option, the formulas are checked in a traditional way. The program will generate a single ISPL input file called ‘input.ispl’, with all seven formulas listed under the Formulae section. Although each formula is checked independently, MCMAS won’t return a result unless all formulas have been verified. Theoretically speaking, if the first formula already returns true, there’s no need to check for others. Thus, one may need to wait longer than the actual time needed to obtain the final result, which leads to a reduction in performance.

WHICH In this option, the program will generate six ISPL input files, namely ‘input.LTL SAT.ispl’, ‘input.LTLSYN_OR1.ispl’, ‘input.LTLSYN_OR2.ispl’, ‘input.LTLSYN_AND.ispl’, ‘input.CTLSTAR_OR1.ispl’ and ‘input.CTLSTAR_OR2.ispl’. Each file contains a unique formula that needs to be verified, except for ‘input.LTLSYN_AND.ispl’ which has Formula number 4 and 5 grouped together. The verification starts with the simplest problem, which is LTL satisfiability checking. If the formula returns true, the result can be directly returned to the user. If the formula returns false, the next simplest problem will be checked. The process continues until the output returns true, or all the formulas have been checked and the output returns false. In this case, MCMAS could be called up to six times. One advantage of this option is that the user doesn’t need to wait for all formulas to be verified, which leads to an improvement in performance. Also, the user can use this option to find out the specific formula that returns true. For instance, the following output could be printed on the screen: ‘The game has a Nash Equilibrium, LTL Satisfiability is satisfied’.

Generally speaking, if performance is not an important factor in the evaluation process, one can always use the default option ‘ALL’ to achieve integrity. Otherwise, for easy and simple games, the user is recommended to use option ‘ALL’; for large and complex games, the user is recommended to use option ‘WHICH’. This statement can be reached with obvious reasons. For a small system, the execution time for MCMAS to check all the formulas is expected to be relatively short (usually less than a second). Thus, it is not worthy to split one single ISPL file into six ISPL files and run MCMAS several times. In contrast, for a complicated system, it might take a plenty of time for MCMAS to check all the formulas. Thus, it is better to split into small input files and check one by one.

4.4.2 The ‘X’ Operator

In ISPL, the section `InitStates` declares all the possible initial states. In theory, we expect MCMAS to randomly pick one as the initial state for both players. However, in reality, MCMAS treat all the possible states listed under `InitStates` section as realistic initial states. This means that it returns true only if the game has a Nash equilibrium for all possible initial states. Consider the following game:

$$G = (\{1, 2\}, \{x, y\}, \{x\}, \{y\}, x, !y)$$

Intuitively, this game should have a Nash equilibrium since the goal for each player only contains their own local variables. There are four possible initial states, and the game is guaranteed to have a Nash equilibrium with initial state $\{x = true, y = false\}$. However, since MCMAS checks for all the possible initial states, the output will be returned as false. To fix this issue, we add the operator ‘X’ in front of the goals for each formula [23]. In LTL, ‘X ’ indicates ‘the next time step’. Since we are not interested in evaluating the game with its initial state, adding ‘X’ allows MCMAS to verify the formulas starting at the next time step after the initial state.

The updated pseudo-code is presented as follows:

Algorithm 4 Two-NE-Satisfiability($AP_1, AP_2, \gamma_1, \gamma_2$)

Require: Adding operator ‘X’

```

1: if  $\ll strategy\_p1 \gg (Player1, strategy\_p1) \ll strategy\_p2 \gg$ 
    $(Player2, strategy\_p2) X (\gamma_1 \wedge \gamma_2)$  then return True
2: if  $\ll strategy\_p1 \gg (Player1, strategy\_p1) \ll strategy\_p2 \gg (Player2, strategy\_p2) X (\gamma_1)$ 
   then return True
3: if  $\ll strategy\_p2 \gg (Player2, strategy\_p2) \ll strategy\_p1 \gg (Player1, strategy\_p1) X (\gamma_2)$ 
   then return True
4: if  $\ll strategy\_p1 \gg (Player1, strategy\_p1) \ll strategy\_p2 \gg (Player2, strategy\_p2) X (!\gamma_2)$ 
   and  $\ll strategy\_p2 \gg (Player2, strategy\_p2) \ll strategy\_p1 \gg (Player1, strategy\_p1) X (!\gamma_1)$ 
   then return True
5: if  $\ll strategy\_p1 \gg (Player1, strategy\_p1) \ll strategy\_p2 \gg (Player2, strategy\_p2) X !\gamma_2$ 
   and  $\ll strategy\_p2 \gg (Player2, strategy\_p2) X \gamma_1$  then return True
6: if  $\ll strategy\_p2 \gg (Player2, strategy\_p2) \ll strategy\_p1 \gg (Player1, strategy\_p1) X !\gamma_1$ 
   and  $\ll strategy\_p1 \gg (Player1, strategy\_p1) X \gamma_2$  then return True
7: return False.

```

Chapter 5

System Evaluation

5.1 Overview

In this chapter, we verify the correctness of the program and evaluate the performance of the system. In section 5.2, we perform the sanity check by writing different test cases and verify the the correctness of outputs. In section 5.3, we conduct several case studies for data collection and result analysis.

Due to the significant amount of testing, we set up an upper limit of 10 minutes for the execution time in the data collection process. Therefore, any data that goes above this limitation is not of interest and will be discarded.

5.2 System Testing

In this section, we discuss two representative examples among all the test cases. In the first example, the game should have a Nash equilibrium, while in the second example, the game should not have a Nash equilibrium.

Example 1 Consider the following iterated Boolean game, which is adapted from a recent paper written by Julian Gutierrez, Paul Harrenstein and Michael Wooldridge [4] :

$$G = (\{1, 2\}, \{x, y\}, \{x\}, \{y\}, GF(x \wedge y), GF(\neg x \wedge \neg y))$$

Here, the game has two players, namely Player1 and Player2. Player1 controls variable x , while Player2 controls variable y . In the paper mentioned above, this game has already been proved to have a Nash equilibrium. Thus, the expected output of our program should be

‘Yes, the game has a Nash equilibrium’

Now let’s set up the user input file and run the program. In this example, the ‘Explore’ feather is also tested.

Figure 5.1 shows the program output with explore option ‘ALL’:

```

3   Please check http://vas.doc.ic.ac.uk/tools/mcmas/ for the latest release.
4   Please send any feedback to <mcmas@imperial.ac.uk>
5   *****
6
7   Command line: ./mcmas examples/iBG/input.ispl
8
9   examples/iBG/input.ispl has been parsed successfully.
10  Global syntax checking...
11  Done
12  Encoding BDD parameters...
13  Building partial transition relation...
14  Building BDD for initial states...
15  Building reachable state space...
16  Checking formulae...
17  Verifying imperfect recall properties...
18  Formula number 1: <<strategy_env>> (Environment, strategy_env) <<strategy_p1>> (Player1,
19  Formula number 2: <<strategy_env>> (Environment, strategy_env) <<strategy_p1>> (Player1,
20  Formula number 3: <<strategy_env>> (Environment, strategy_env) <<strategy_p2>> (Player2,
21  Formula number 4: <<strategy_env>> (Environment, strategy_env) <<strategy_p1>> (Player1,
22  Formula number 5: <<strategy_env>> (Environment, strategy_env) <<strategy_p2>> (Player2,
23  Formula number 6: <<strategy_env>> (Environment, strategy_env) <<strategy_p1>> (Player1,
24  Formula number 7: <<strategy_env>> (Environment, strategy_env) <<strategy_p2>> (Player2,
25  done, 7 imperfect recall formulae successfully read and checked
26  execution time = 0.021
27  number of reachable states = 4
28  BDD memory in use = 8989648
29  -----
30  1471866403.91:finishing MCMAS...
31  -----
32  Execution Time:0.073292016983 seconds
33  -----
34  The game has a Nash Equilibrium
35  -----

```

Figure 5.1: MCMAS output with option ‘ALL’

From the above diagram, we can see that the ISPL file generated by the program has been parsed successfully by MCMAS. Since the option ‘ALL’ is used, all the formulas are defined in the same input file and checked independently. At line 34, the output of the program states that ‘The game has a Nash Equilibrium’, which is the output we expect to see.

Figure 5.2 shows the program output with explore option ‘WHICH’:

```

 7   Please check http://vas.doc.ic.ac.uk/tools/mcmas/ for the latest release.
 8   Please send any feedback to <mcmas@imperial.ac.uk>
 9   ****
10
11  Command line: ./mcmas examples/iBG/input_LTLSYN_AND.ispl
12
13  examples/iBG/input_LTLSYN_AND.ispl has been parsed successfully.
14  Global syntax checking...
15  Done
16  Encoding BDD parameters...
17  Building partial transition relation...
18  Building BDD for initial states...
19  Building reachable state space...
20  Checking formulae...
21  Verifying imperfect recall properties...
22    Formula number 1: <<strategy_env>> (Environment, strategy_env) <<strategy_p1>> (Player
23    Formula number 2: <<strategy_env>> (Environment, strategy_env) <<strategy_p2>> (Player
24  done, 2 imperfect recall formulae successfully read and checked
25  execution time = 0.009
26  number of reachable states = 4
27  BDD memory in use = 8989648
28  -----
29  1471865903.88:finishing MCMAS...
30  -----
31  Execution Time:0.0349900722504 seconds
32  -----
33  The game has a Nash Equilibrium
34  LTL SYN(AND) is satisfied
35  -----

```

Figure 5.2: MCMAS output with option ‘WHICH’

From the above diagram, we can see that the program generates an ISPL file for each individual formula. With the ‘WHICH’ option, the user can get additional information on the specific formula that returns true in the checking process. At Line 34, the output of the program states that ‘The game has a Nash Equilibrium, LTL SYN(AND) is satisfied’. This implies that the formula that checks for the LTL synthesis returns true. Thus, we can conclude that the output is as expected.

Example 2 Now let’s consider a real world application: the matching pennies game. The game has two players. To play the game, each player picks one side (either head or tail) of the penny simultaneously and compare their choices. If the penny matches, player1 wins. Otherwise player2 wins. We can formalize this game into the following iterated Boolean game:

$$G = (\{1, 2\}, \{x, y\}, \{x\}, \{y\}, (x \wedge y) \text{ or } (\neg x \wedge \neg y), (\neg x \wedge y) \text{ or } (x \wedge \neg y))$$

Here, player1 controls variable x, and player2 controls variable y. We define x,y as boolean variables, with true as head, and false as tail. Player1’s goal states that ‘both players choose heads or tails’, while player2’s goal states that ‘one player chooses head, and the other chooses tail’. Remember that in this project, we only consider the use of pure-strategies. Since the matching pennies game doesn’t have pure-strategy Nash equilibrium [24], the expected output of our program should be:

‘This game does not have a Nash Equilibrium’

Now let’s set up the user input file and run the program. In this example, explore option ‘ALL’ is used for simplicity. Figure 5.3 shows a screenshot of the program output:

```

7   Please check http://vas.doc.ic.ac.uk/tools/mcmas/ for the latest release.
8   Please send any feedback to <mcmas@imperial.ac.uk>
9   ****
10
11  Command line: ./mcmas examples/iBG/input.ispl
12
13  examples/iBG/input.ispl has been parsed successfully.
14  Global syntax checking...
15  Done
16  Encoding BDD parameters...
17  Building partial transition relation...
18  Building BDD for initial states...
19  Building reachable state space...
20  Checking formulae...
21  Verifying imperfect recall properties...
22    Formula number 1: <<strategy_env>> (Environment, strategy_env) <<strategy_p1>> (Player1,
23    Formula number 2: <<strategy_env>> (Environment, strategy_env) <<strategy_p1>> (Player1,
24    Formula number 3: <<strategy_env>> (Environment, strategy_env) <<strategy_p2>> (Player2,
25    Formula number 4: <<strategy_env>> (Environment, strategy_env) <<strategy_p1>> (Player1,
26    Formula number 5: <<strategy_env>> (Environment, strategy_env) <<strategy_p2>> (Player2,
27    Formula number 6: <<strategy_env>> (Environment, strategy_env) <<strategy_p1>> (Player1,
28    Formula number 7: <<strategy_env>> (Environment, strategy_env) <<strategy_p2>> (Player2,
29  done, 7 imperfect recall formulae successfully read and checked
30  execution time = 0.025
31  number of reachable states = 4
32  BDD memory in use = 8989648
33  -----
34  1471867884.49:finishing MCMAS...
35  -----
36  Execution Time:0.13257598877 seconds
37  -----
38  The game doesn't have a Nash Equilibrium
39  -----

```

Figure 5.3: MCMAS output for the Matching Pennies Game

At line 38, the output of the program states that ‘The game doesn’t have a Nash Equilibrium’, which is result we expect to see.

Summary In this section, we give two example test cases to verify the correctness of the program. From the results obtained above, we can conclude that the program has been properly designed to function correctly in solving the existence of Nash equilibrium problem. Since there is no existing benchmarks that can be used for our implementation, we will shift our focus to the performance evaluation in the next section.

5.3 Performance Evaluation

In this section, we conduct several case studies to evaluate the performance of our program.

5.3.1 Case Study 1: Simple goals, vary the number of variables

In this case, we keep simple goals, and gradually increase the number of variables controlled by each player. The goals used are defined as follows:

$$\text{goal1} = x_0 ; \text{goal2} = !y_0$$

Here, x_0 is the variable controlled by player1, and y_0 is the variable controlled by player2. The following table summarizes the execution time where each player can control up to 10 variables:

# of variables		Player1									
		1	2	3	4	5	6	7	8	9	10
Player2	1	0.0103s									
	2	0.0102s	0.0117s								
	3	0.0123s		0.0225s							
	4	0.0210s	0.0302s		0.2980s						
	5	0.0599s		0.2419s		8.2808s					
	6	0.2977s	0.4352s		9.8658s		173.4656s				
	7	2.046s		11.6758s		201.0617s		>10 min			
	8	9.7875s	18.4468s		217.0709s		>10 min		>10 min		
	9	56.2882s		256.6434s		>10 min		>10 min		>10 min	
	10	270.1404s	>10 min		>10 min		>10 min		>10 min		>10 min

To read this table, we can select the number of variables controlled by each player, and search for the intersection value. For instance, with player1 controls 3 variables and player2 controls 5 variables, the execution times is read as 0.2491s. Here, the unit ‘s’ is in second. Since the goals for both players are equally simple, the execution time for player1 controls x variables and player2 controls y variables should be similar as the other way around. Thus, we can eliminate half of the test cases. Moreover, ‘>10 min’ indicates that the execution time is more than 10 minutes and is out of scope.

Result Analysis:

We can analyze the results in two different settings. For 1:n setting (player1 controls 1 variable, player2 controls n variables), the program can take up to 10 variables, with an average waiting time of 5 minutes. If n is greater than 10, we have to wait for more than 10 minutes. For m:n setting (player1 controls m variable, player2 controls n variables), the program can take up to 12 variables in total, with an average waiting time of 2 minutes. It is obvious to see that the closer the number of variables controlled by each player, the better the performance. To give an example, suppose there are 10 variables in total. For the 3:7 setting (player1 controls 3 variable, player2 controls 7 variables), the execution time is 11.6758s. For the 5:5 setting (both players control 5 variables), the execution time is 8.2808s.

5.3.2 Case Study 2: Few variables, vary the complexity of goals

5.3.2.1 Goals with F only

In this section, we test for program reachability, with goals that only contain the unary operator ‘F’.

We start with two variables:

Player1	Player2	Execution Time
F x_0	F $!y_0$	0.014465809s
F x_0 and y_1	F $!y_0$ and x_1	0.013870955s
F (x_0 and y_1) or $!y_0$	F ($!y_0$ and x_0) or $!x_1$	0.113681078s
F (y_0 and x_1) or F $!y_0$ and x_1	F ($!y_0$ and x_1) or F y_1 and $!x_0$	0.136762857s

Table 5.1: Goal with F only: Two variables

Tables 5.1 presents the program execution time where player1 controls $\{x_0, x_1\}$ and player2 controls $\{y_0, y_1\}$. For each test case, we gradually increase the complexity of both goals. The result shows that even with the most complicated goals (refer to last row in the table), the execution time is less than 0.2 second.

Now let's increase the number of variables to three:

Player1	Player2	Execution Time
F x_0 and y_1	F $!y_0$ and x_1	0.325232029s
F (x_0 and y_1) or $!y_2$	F ($!y_0$ and x_1) or $!x_2$	>10 min

Table 5.2: Goal with F only: Three variables

Tables 5.2 presents the program execution time where player1 controls $\{x_0, x_1, x_2\}$ and player2 controls $\{y_0, y_1, y_2\}$. In the first row, each goal contains two variables. The execution time is around 0.4s, which is acceptable. In the second row, each goal contains three variables and the execution time is more than 10 minutes. Thus, it can be concluded that the performance of the program will be degraded with more than three variables for each player.

Result Analysis:

Based on the experimental data, we can conclude that the program achieves excellent performance with a maximum of two variables per player, and achieve good performance with three variables but simple goals. However, with three or more variables and complex goals, it will become very time consuming.

5.3.2.2 Goals with G only

In this section, we test for program safeness, with goals that only contain the unary operator 'G'.

We start with two variables:

Player1	Player2	Execution Time
G x_0	G $!y_0$	0.008846998s
G x_0 and y_1	G $!y_0$ and x_1	0.008055925s
G (x_0 and y_1) or $!y_0$	G ($!y_0$ and x_0) or $!x_1$	0.008934021s
G (y_0 and x_1) or G $!y_0$ and x_1	G ($!y_0$ and x_1) or G y_1 and $!x_0$	0.011085987s

Table 5.3: Goal with G only: Two variables

Tables 5.3 presents the program execution time where player1 controls $\{x_0, x_1\}$ and player2 controls $\{y_0, y_1\}$. It is obvious to see that the execution time is very short, with an average of less than 0.1 second.

Now let's increase the number of variables to three:

Player1	Player2	Execution Time
G x_0 and y_1	G $!y_0$ and x_1	0.639224052s
G (x_0 and y_1) or ($!y_2$ and x_2)	G ($!y_0$ and x_0) or G(y_2 and $!x_1$)	1.634618044s
G (x_0 and y_1) or ($!y_2$ and x_2) and (x_0 or y_0)	G ($!y_0$ and x_0) or G(y_2 and $!x_1$) and (x_2 or y_1)	2.00779295s

Table 5.4: Goal with G only: Three variables

Tables 5.4 presents the program execution time where player1 controls $\{x_0, x_1, x_2\}$ and player2 controls $\{y_0, y_1, y_2\}$. Compare with the two-variable case, the execution time is longer, with an average of 2 seconds in the worst case scenario. Since the worst case execution time is still under scope, we increase the number of variables to four:

Player1	Player2	Execution Time
G x_0	G $!y_0$	>10 min

Table 5.5: Goal with G only: Four variables

Tables 5.5 presents the program execution time where player1 controls $\{x_0, x_1, x_2, x_3\}$ and player2 controls $\{y_0, y_1, y_2, y_3\}$. We can see that even with the simplest goals, the program becomes very time consuming and the execution time is more than 10 minutes.

Result Analysis:

Based on the experimental data, we can conclude that the program achieves excellent performance with a maximum of three variables per player. However, with four or more variables and even the simplest goals, it will become very time consuming.

5.3.2.3 Goals with U only

In this section, we test the system performance with goals that only contain binary operator ‘U’.

We start with two variables:

Player1	Player2	Execution Time
$x_0 \text{ U } y_0$	$!x_0 \text{ U } !y_0$	0.008466005s
$(x_0 \text{ U } y_1) \text{ or } (!y_0 \text{ U } x_1)$	$(!x_0 \text{ U } !y_1) \text{ or } (y_0 \text{ U } x_1)$	0.009564161s
$((x_0 \text{ U } y_0) \text{ and } !y_1) \text{ or } (y_1 \text{ U } !x_1)$	$((!x_0 \text{ U } !y_0) \text{ and } x_1) \text{ or } (!y_1 \text{ U } x_1)$	0.009805202s

Table 5.6: Goal with U only: Two variables

Tables 5.6 presents the program execution time where player1 controls $\{x_0, x_1\}$ and player2 controls $\{y_0, y_1\}$. The result shows that the average execution time is less than 0.1 second.

Now let’s increase the number of variables to three:

Player1	Player2	Execution Time
$x_0 \text{ U } y_0$	$!x_0 \text{ U } !y_0$	1.003838062s
$(x_0 \text{ U } y_1) \text{ or } (!y_0 \text{ U } x_1)$	$(!x_0 \text{ U } !y_1) \text{ or } (y_0 \text{ U } x_1)$	1.592059851
$((x_0 \text{ U } y_0) \text{ and } !y_1) \text{ or } (y_1 \text{ U } !x_1 \text{ and } x_2)$	$((!x_0 \text{ U } !y_0) \text{ and } x_1) \text{ or } (!y_1 \text{ U } x_2 \text{ and } y_2)$	>10 min

Table 5.7: Goal with U only: Three variables

Tables 5.7 presents the program execution time where player1 controls $\{x_0, x_1, x_2\}$ and player2 controls $\{y_0, y_1, y_2\}$. We can see that with complex goals, the execution time is more than 10 minutes.

Result Analysis:

Based on the experimental data, we can conclude that the program achieves good performance with a maximum of three variables per player. The performance will be greatly reduced with four or more variables.

5.3.2.4 Goal with G F U X

In this section, we test the program with goals that are expressed in full LTL formulas.

We start with two variables:

Player1	Player2	Execution Time
$G F (x_0 U y_0)$	$G F (!x_0 U !y_0)$	0.015619993s
$G F (x_0 U y_1) \text{ and } X(!y_0 U x_1)$	$G F (!x_0 U !y_1) \text{ or } X(y_1 U x_1)$	0.030667782s
$G F ((x_1 U y_0) \text{ and } !y_1) \text{ or } (X y_1 U !x_0)$	$G F (!x_1 U !y_0) \text{ and } x_0 \text{ or } X (!y_1 U x_0)$	0.127990961s

Table 5.8: Goal expressed in full LTL: Two variables

Tables 5.8 presents the program execution time where player1 controls $\{x_0, x_1\}$ and player2 controls $\{y_0, y_1\}$. The result shows that the average execution time is less than 0.1 second, with a value of 0.13 second in the worst case.

Now let's increase the number of variables to three:

Player1	Player2	Execution Time
$G F (x_0 U y_0)$	$G F (!x_0 U !y_0)$	296.11058904s
$G F (x_0 U y_1) \text{ and } X(!y_0 U x_1)$	$G F (!x_0 U !y_1) \text{ or } X(y_2 U x_1)$	>10 min

Table 5.9: Goal expressed in full LTL: Three variables

Tables 5.9 presents the program execution time where player1 controls $\{x_0, x_1, x_2\}$ and player2 controls $\{y_0, y_1, y_2\}$. The result shows that for simple goals, the execution time is around 5 minutes. For more complicate goals, the execution time is more than ten minutes.

Result Analysis:

Based on the experimental data, we can conclude that the program achieves good performance with a maximum of two variables per player. However, with three or more variables, the performance will be greatly reduced.

5.3.2.5 Summary

In section 5.3.2, we evaluate the program performance by limiting the number of variables and varying the complexity of goals. The following conclusion can be drawn from the experimental results:

- ① Performance ranking from high to low: $G > F > U > \text{Full LTL formula}$
- ② In general, the program achieve excellent performance when each player controls no more than two variables. The average execution time is around 0.1s.
- ③ With three or more variables and complex goals, the program becomes time consuming, which leads a great reduction in performance.

5.3.3 Case Study 3 : ALL vs WHICH

In this section, we compare the performance with different explore options: ‘ALL’ and ‘WHICH’.

For option ‘ALL’, we can reuse the results presented in section 5.3.1.

Figure 5.4 shows the testing results:

# of variables		Player1									
		1	2	3	4	5	6	7	8	9	10
Player2	1	0.0103s									
	2	0.0102s	0.0117s								
	3	0.0123s		0.0225s							
	4	0.0210s	0.0302s		0.2980s						
	5	0.0599s		0.2419s		8.2808s					
	6	0.2977s	0.4352s		9.8658s		173.4656s				
	7	2.046s		11.6758s		201.0617s		>10 min			
	8	9.7875s	18.4468s		217.0709s		>10 min		>10 min		
	9	56.2882s		256.6434s		>10 min		>10 min		>10 min	
	10	270.1404s	>10 min		>10 min		>10 min		>10 min		>10 min

Figure 5.4: Execution Time with option ‘ALL’

For option ‘WHICH’, we use the same test cases as in option ‘ALL’.

Figure 5.5 shows the testing results:

# of variables		Player1									
		1	2	3	4	5	6	7	8	9	10
Player2	1	0.0668s									
	2	0.0099s	0.0095s								
	3	0.0101s		0.0121s							
	4	0.0119s	0.0134s		0.0635s						
	5	0.01993		0.0645s		1.1060s					
	6	0.0566s	0.0861s		1.3069s		173.4656s				
	7	0.2652s		1.6514s		29.3588s		>10 min			
	8	1.1148s	2.2099s		36.5159s		>10 min		>10 min		
	9	6.9279s		42.4042s		>10 min		>10 min		>10 min	
	10	34.8204s	60.5101s		>10 min		>10 min		>10 min		>10 min

Figure 5.5: Execution Time with option ‘WHICH’

Result Analysis:

With few variables, there’s not much difference in the execution time between the two options. However, as the number of variables increases, we can see a great improvement in performance by using the option ‘WHICH’. To give an example, consider the case where player1 controls 5 variables and player2 controls 7 variables. With option ‘ALL’, the execution time is 201.0617s, while with option ‘WHICH’, the execution time is only 29.3588s. Based on the comparison of the two groups of data, we can conclude that the option ‘WHICH’ should be used when five or more variables are controlled by each player.

Chapter 6

Conclusion

6.1 Dissertation Remarks

In this dissertation, we introduced the essential background information needed for the project and discussed the algorithm used. Besides, we explained the detailed implementation process, and conducted several experiments for system evaluation.

The scope of the project is to solve the ‘Existence of Nash Equilibria’ problem in two-player iterated Boolean games. The algorithm used is adapted from a recent paper written by Julian Gutierrez, Paul Harrenstein, and Michael Wooldridge [5]. It was shown that checking the existence of Nash equilibrium can be reduced to a combination of temporal logic satisfiability and synthesis problems for two-player games. Therefore, this project aims to implement the decision procedure based on the existing algorithm to achieve automated analysis of Nash equilibria in practice.

On the implementation side, MCMAS [20] is used as the model checker tool to solve the satisfiability and synthesis problems for temporal logics. The program is designed in a user-friendly way, which allows the user to construct different types of games by modifying the user input file (input.ibg). In terms of functional capabilities, the ‘Explore’ feather is designed to provide more flexibility for the user to choose between alternative ways of checking the formulas. The complete source code has been uploaded on Github and is ready to use. Detailed instructions can be found in Appendix B.

A complete testing plan is proposed for system validation and performance evaluation. The program has been proven to work properly in providing a platform for the user to interact with the system. Besides, the program has been proven to be successful in solving the ‘existence of Nash Equilibria’ problem.

Based on the testing results in performance evaluation, we can conclude that the program achieves excellent performance in checking the existence of Nash equilibrium for two-player game with a maximum of two variables controlled by each player. The performance will be degraded with three or more variables and complex goals. Remember that MCMAS adapts memoryless version in solving the synthesis problem due to its high computational complexity. Thus, we can see that checking the existence of Nash equilibrium is indeed a very hard problem. In solving such problem, performance is the main issue that needs to be taken into consideration, even for small specification.

In summary, the goal of the project has been accomplished successfully, and the implemented system has met all the design specifications.

6.2 Achievements

In this project, we implement a program to achieve automated analysis in solving the ‘Existence of Nash Equilibria’ problem. Compare with other open source software, our system has the following advantages:

- The system adapts the algorithm that reduces the current problem into simpler sub-problems, which leads to an improvement in performance.
- The system is simple and easy to operate, even for users with little or no experience in python.
- The system is implemented to be flexible for any changes in the design requirements.
- The system achieves excellent performance in solving the ‘Existence of Nash Equilibria’ problem for small specifications.

Generally speaking, the user is recommended to use the system to check the existence of Nash Equilibria for small-scale concurrent two-player games.

6.3 Recommendations for Future Work

The project encountered several design challenges during the implementation process. In the initial plan, we aim to find pre-developed online libraries for solving the LTL satisfiability problem and LTL/CTL* synthesis problems. However, after doing some research, we did not find any available libraries that can be ready to use to solve the CTL* synthesis problem. Thus, we decide to use MCMAS as a substitution, which in turn cause a scarification in performance.

Since the current design only works for two-player games, more research can be done on how to extend the algorithm so that it can be used for n-player games. To improve the performance of the current system, one can dig into the MCMAS source code, and see if we can eliminate any redundant components that are not used while running the software. Besides, several design extensions can be added to the current system. For instance, apart from solving the Non-Emptiness problem, the program can be optimized to solve other kinds of decision problems, such as the ENASH and ANASH problems, as studied in [6].

Appendix A

Python Source Code

A.1 sourcode.py

```
#!/usr/bin/python
import itertools
import io
import os
import subprocess
import time
import re

# Helper function
# Write comments in ISPL file
def writeComment(fo,var1,var2,goal1,goal2):
    fo.write("-- Input iBG file" + "\n")
    fo.write("-- Player1: " + ", ".join(var1) + "\n")
    fo.write("-- Player2: " + ", ".join(var2) + "\n")
    fo.write("-- Goal1: " + goal1 + "\n")
    fo.write("-- Goal2: " + goal2 + "\n")
    fo.write("\n"*2)

# Generate dummy enviroment
def createEnvironment(fo):
    fo.write("Agent Environment\n")
    fo.write("  Obsvars:\n" + "      dummy : boolean;\n" + "  end\n" + "  Obsvars\n")
    fo.write("  Actions = {none};\n")
    fo.write("  Protocol:\n" + "      Other : {none};\n" + "  end\n" + "  Protocol\n")
    fo.write("  Evolution:\n" + "      dummy = true if Action=none\n" + "      ;\n" + "  end Evolution\n")
    fo.write("end Agent\n")
    fo.write("\n"*2)
```

```

# Generate player
def createPlayer(fo,var,num):
    fo.write("Agent Player1\n") if num == "1" else fo.write("
        Agent Player2\n")
    # action list
    seqList = ["".join(seq) for seq in itertools.product("01",
        repeat = len(var))]
    actionList = ["ac" + seq for seq in seqList] # store actions
    # variables
    fo.write("  Vars:\n")
    for i in var:
        fo.write("    " + i + " : boolean;\n")
    fo.write("  end Vars\n")
    # actions
    fo.write("  Actions = {" + ", ".join(actionList) + "};\n")
    # protocols
    fo.write("  Protocol:\n")
    fo.write("    Other : {" + ", ".join(actionList) + "};\n")
    fo.write("  end Protocol\n")
    # evolution
    fo.write("  Evolution:\n")
    for i in range(len(seqList)):
        fo.write("    ")
        for j in range(len(var)):
            fo.write("(" + var[j] + " = " + str(bool(int(seqList[
                i][j]))).lower() + ")")
            if (j == len(var) - 1): break
            fo.write(" and ")
        fo.write(" if Action=" + actionList[i] + ";\n")
    fo.write("  end Evolution\n")

    fo.write("end Agent\n")
    fo.write("\n"*2)
# Generate Evaluation
def createEvaluation(fo,var1,var2):
    fo.write("Evaluation\n")
    for item in var1:
        fo.write("  " + item + " if " + "Player1." + item + " =
            true;\n")
    for item in var2:
        fo.write("  " + item + " if " + "Player2." + item + " =
            true;\n")
    fo.write("end Evaluation\n")
    fo.write("\n"*2)

```

```

# Generate Initial states
def createInitStates(fo,var1,var2):
    fo.write("InitStates\n")
    fo.write("  (Environment.dummy = true) and\n")

    for item in var1:
        fo.write("  ((Player1." + item + " = true) or (Player1."
            + item + " = false)) and\n")
    for i, item in enumerate(var2):
        fo.write("  ((Player2." + item + " = true) or (Player2."
            + item + " = false))")
        if (i == len(var2) - 1):
            fo.write(";\n")
            break
        else:
            fo.write(" and\n")
    fo.write("end InitStates\n")
    fo.write("\n"*2)

# Generate formula for LTLSAT
def createFormulae_LTLSAT(fo,goal1,goal2):
    fo.write("Formulae\n")
    fo.write("  — LTL SAT\n")
    fo.write("  <<strategy_env>> (Environment, strategy_env) (\n")
    fo.write("    ( <<strategy_p1>> (Player1, strategy_p1) <<
        strategy_p2>> (Player2, strategy_p2) ")
    fo.write(" X " + "(" + goal1 + ") and (" + goal2 + ") ) )\n")
    fo.write("  );\n")
    fo.write("end Formulae\n")

# Generate formula for LTLSYN_OR1
def createFormulae_LTLSYN_OR1(fo,goal1,goal2):
    fo.write("Formulae\n")

    fo.write("  — LTL SYN (OR1)\n")
    fo.write("  <<strategy_env>> (Environment, strategy_env) (\n")
    fo.write("    ( <<strategy_p1>> (Player1, strategy_p1) [[
        strategy_p2]] (Player2, strategy_p2) ")
    fo.write(" X " + "(" + goal1 + ") )\n")
    fo.write("  );\n")
    fo.write("end Formulae\n")

```

```

# Generate formula for LTLSYN_OR2
def createFormulae_LTLSYN_OR2(fo,goal1,goal2):
    fo.write("Formulae\n")
    fo.write("  — LTL SYN (OR2)\n")
    fo.write("  <<strategy-env>> (Environment, strategy-env) (\n")
    fo.write("    ( <<strategy-p2>> (Player2, strategy-p2) [[\n")
    fo.write("      strategy-p1]] (Player1, strategy-p1) ")
    fo.write(" X " + "(" + goal2 + ") )\n")
    fo.write("  );\n")
    fo.write("end Formulae\n")

# Generate formula for LTLSYN_AND
def createFormulae_LTLSYN_AND(fo,goal1,goal2):
    fo.write("Formulae\n")
    fo.write("  — LTL SYN (AND)\n")
    fo.write("  <<strategy-env>> (Environment, strategy-env) (\n")
    fo.write("    ( <<strategy-p1>> (Player1, strategy-p1) [[\n")
    fo.write("      strategy-p2]] (Player2, strategy-p2) ")
    fo.write(" X " + "(!(" + goal2 + ") ) )\n")
    fo.write("  );\n")

    fo.write("  <<strategy-env>> (Environment, strategy-env) (\n")
    fo.write("    ( <<strategy-p2>> (Player2, strategy-p2) [[\n")
    fo.write("      strategy-p1]] (Player1, strategy-p1) ")
    fo.write(" X " + "(!(" + goal1 + ") ) )\n")
    fo.write("  );\n")
    fo.write("end Formulae\n")

# Generate formula for CTLStarSYN_OR1
def createFormulae_CTLStarSYN_OR1(fo,goal1,goal2):
    fo.write("Formulae\n")
    fo.write("  — CTL* SYN (OR1)\n")
    fo.write("  <<strategy-env>> (Environment, strategy-env) (\n")
    fo.write("    ( <<strategy-p1>> (Player1, strategy-p1) ")
    fo.write("( [[strategy-p2]] (Player2, strategy-p2) " + " X "
    + "(!(" + goal2 + ") )")
    fo.write(" and ")
    fo.write("<<strategy-p2>> (Player2, strategy-p2) " + " X " +
    "(" + goal1 + ") ) )\n")
    fo.write("  );\n")
    fo.write("end Formulae\n")

```

```

# Generate formula for CTLStarSYN_OR2
def createFormulae_CTLStarSYN_OR2(fo,goal1,goal2):
    fo.write("Formulae\n")
    fo.write("  — CTL* SYN (OR2)\n")
    fo.write("  <<strategy-env>> (Environment, strategy-env) (\n")
    fo.write("    ( <<strategy-p2>> (Player2, strategy-p2) ")
    fo.write("( [[strategy-p1]] (Player1, strategy-p1) " + " X "
        + "(!(" + goal1 + ") )")
    fo.write(" and ")
    fo.write("<<strategy-p1>> (Player1, strategy-p1) " + " X " +
        "(" + goal2 + ") ) )\n")
    fo.write(" );\n")
    fo.write("end Formulae\n")

# Generate all formulas
def createFormulae(fo,goal1,goal2):
    fo.write("Formulae\n")
    # LTL SAT
    fo.write("  — LTL SAT\n")
    fo.write("  <<strategy-env>> (Environment, strategy-env) (\n")
    fo.write("    ( <<strategy-p1>> (Player1, strategy-p1) <<
        strategy-p2>> (Player2, strategy-p2) ")
    fo.write(" X " + "(" + goal1 + ") and (" + goal2 + ") ) )\n")
    fo.write(" )")
    fo.write(" );\n")
    fo.write("\n")

    # LTL SYN1
    fo.write("  — LTL SYN\n")
    fo.write("  <<strategy-env>> (Environment, strategy-env) (\n")
    fo.write("    ( <<strategy-p1>> (Player1, strategy-p1) [[
        strategy-p2]] (Player2, strategy-p2) ")
    fo.write(" X " + "(" + goal1 + ") )\n")
    fo.write(" );\n")

    fo.write("  <<strategy-env>> (Environment, strategy-env) (\n")
    fo.write("    ( <<strategy-p2>> (Player2, strategy-p2) [[
        strategy-p1]] (Player1, strategy-p1) ")
    fo.write(" X " + "(" + goal2 + ") )\n")
    fo.write(" );\n")
    fo.write("\n")

    # LTL SYN2
    fo.write("  — LTL SYN\n")
    fo.write("  <<strategy-env>> (Environment, strategy-env) (\n")
    fo.write("    ( <<strategy-p1>> (Player1, strategy-p1) [[
        strategy-p2]] (Player2, strategy-p2) ")

```

```

fo.write(" X " + "(!(" + goal2 + ") ) )\n" )
fo.write(" );\n")

fo.write(" <<strategy-env>> (Environment, strategy-env) (\n")
fo.write("      ( <<strategy-p2>> (Player2, strategy-p2) [[
      strategy-p1]] (Player1, strategy-p1) ")
fo.write(" X " + "(!(" + goal1 + ") ) )\n")
fo.write(" );\n")
fo.write("\n")

# CTL* SYN
fo.write(" — CTL* SYN\n")
fo.write(" <<strategy-env>> (Environment, strategy-env) (\n")
fo.write("      ( <<strategy-p1>> (Player1, strategy-p1) ")
fo.write("( [[strategy-p2]] (Player2, strategy-p2) " + " X "
      + "(!(" + goal2 + ") )")
fo.write(" and ")
fo.write("<<strategy-p2>> (Player2, strategy-p2) " + " X " +
      "(" + goal1 + ") ) )\n")
fo.write(" );\n")

fo.write(" <<strategy-env>> (Environment, strategy-env) (\n")
fo.write("      ( <<strategy-p2>> (Player2, strategy-p2) ")
fo.write("( [[strategy-p1]] (Player1, strategy-p1) " + " X "
      + "(!(" + goal1 + ") )")
fo.write(" and ")
fo.write("<<strategy-p1>> (Player1, strategy-p1) " + " X " +
      "(" + goal2 + ") ) )\n")
fo.write(" );\n")
fo.write("\n")
fo.write("end Formulae\n")

# Generate formula for SL
def createFormulaeSL(fo, goal1, goal2):
    fo.write("Formulae\n")
    fo.write(" <<strategy-env>> (Environment, strategy-env) (\n")
    fo.write("      <<strategy-p1>> <<strategy-p2>> (Player1,
      strategy-p1) (Player2, strategy-p2) ")
    fo.write("( (" + goal1 + ") or [[alt_strategy-p1]] (Player1,
      alt_strategy-p1)" + " !(" + goal1 + ") )")
    fo.write(" and ")
    fo.write("( (" + goal2 + ") or [[alt_strategy-p2]] (Player2,
      alt_strategy-p2)" + " !(" + goal2 + ") )")
    fo.write(" );\n")
    fo.write("end Formulae\n")

```

```

# Run MCMAS
def runMCMAS(filename):
    start_time = time.time()
    print("%s:running MCMAS..." % start_time)
    print("_____")

    # Call mcmas
    os.chdir("../..") # change directory
    output = os.popen("./mcmas examples/iBG/" + filename).read()

    # Set flag
    done = False

    # track excution time
    print output
    print("_____")
    print("%s:finishing MCMAS..." % time.time())
    print("_____")
    end_time = time.time() - start_time
    print("Execution Time:%s seconds " % end_time)
    print("_____")

    f = open('examples/iBG/excutiontime.ibg','a+')
    f.write("Execution Time:%s seconds\n" % end_time)
    f.close()

#identidy filename
if filename == "input.ispl":
    # Parse output
    result1 = re.search("Formula number 1(.*?)model", output)
    result2 = re.search("Formula number 2(.*?)model", output)
    result3 = re.search("Formula number 3(.*?)model", output)
    result4 = re.search("Formula number 4(.*?)model", output)
    result5 = re.search("Formula number 5(.*?)model", output)
    result6 = re.search("Formula number 6(.*?)model", output)
    result7 = re.search("Formula number 7(.*?)model", output)

    if (("is TRUE" in result1.group(1)) or ("is TRUE" in
        result2.group(1)) or
        ("is TRUE" in result3.group(1)) or ("is TRUE" in
            result6.group(1)) or
        ("is TRUE" in result7.group(1))):
        # raise flag
        done = True
        # print result
        print "The game has a Nash Equilibrium "
    elif ("is TRUE" in result4.group(1)) and ("is TRUE" in

```

```

        result5.group(1)):
        # raise flag
        done = True
        # print result
        print "The game has a Nash Equilibrium "
    else:
        print "The game doesn't have a Nash Equilibrium "
elif filename == "input_LTLSAT.ispl":
    if "is TRUE in the model" in output:
        # raise flag
        done = True
        # print result
        print "The game has a Nash Equilibrium "
        print "LTL SAT is satisfied"
elif filename == "input_LTLSYN_OR1.ispl":
    if "is TRUE in the model" in output:
        # raise flag
        done = True
        # print result
        print "The game has a Nash Equilibrium "
        print "LTL SYN(OR1) is satisfied"
elif filename == "input_LTLSYN_OR2.ispl":
    if "is TRUE in the model" in output:
        # raise flag
        done = True
        # print result
        print "The game has a Nash Equilibrium "
        print "LTL SYN(OR2) is satisfied"
elif filename == "input_LTLSYN_AND.ispl":
    if "is FALSE in the model" not in output:
        # raise flag
        done = True
        # print result
        print "The game has a Nash Equilibrium "
        print "LTL SYN(AND) is satisfied"
elif filename == "input_CTLSTAR_OR1.ispl":
    if "is TRUE in the model" in output:
        # raise flag
        done = True
        # print result
        print "The game has a Nash Equilibrium "
        print "CTL* SYN(OR1) is satisfied"
elif filename == "input_CTLSTAR_OR2.ispl":
    if "is TRUE in the model" in output:
        # raise flag
        done = True
        # print result
        print "The game has a Nash Equilibrium "

```



```

        print "CTL* SYN(OR2) is satisfied"
    else:
        print "The game doesn't have a Nash Equilibrium "

print("_____")

os.chdir("examples/iBG/") # change directory back

return done

```

```

# Generate single ISPL file, used in option 'ALL'
def translateAll(var1,var2,goal1,goal2):
    print(str(time.time()) + ":generating .ispl file...")
    print("_____")
    # Translate to .ispl file
    filename = "input.ispl"
    fo = open(filename, "w")
    writeComment(fo,var1,var2,goal1,goal2)
    createEnvironment(fo)
    createPlayer(fo,var1,"1")
    createPlayer(fo,var2,"2")
    createEvaluation(fo,var1,var2)
    createInitStates(fo,var1,var2)
    createFormulae(fo,goal1,goal2)
    fo.close()
    # Run MCMAS
    runMCMAS(filename)

def translateSL(var1,var2,goal1,goal2):
    print(str(time.time()) + ":generating .ispl file...")
    print("_____")
    # Translate to .ispl file
    filename = "input.ispl"
    fo = open(filename, "w")
    writeComment(fo,var1,var2,goal1,goal2)
    createEnvironment(fo)
    createPlayer(fo,var1,"1")
    createPlayer(fo,var2,"2")
    createEvaluation(fo,var1,var2)
    createInitStates(fo,var1,var2)
    createFormulaeSL(fo,goal1,goal2)
    fo.close()
    # Run MCMAS
    runMCMAS(filename)

```

Generate ISPL file for each formula, used in option 'WHICH'

```
def translateEach(var1,var2,goal1,goal2):  
    # Translate to 4 .ispl file and run one by one
```

```
    ### LTL SAT
```

```
    filename = "input_LTLSAT.ispl"  
    fo = open(filename, "w")  
    writeComment(fo,var1,var2,goal1,goal2)  
    createEnvironment(fo)  
    createPlayer(fo,var1,"1")  
    createPlayer(fo,var2,"2")  
    createEvaluation(fo,var1,var2)  
    createInitStates(fo,var1,var2)  
    createFormulae_LTLSAT(fo,goal1,goal2)  
    fo.close()
```

```
    # Run MCMAS
```

```
    done = runMCMAS(filename)  
    if done: return
```

```
    ### LTL SYN (OR1)
```

```
    filename = "input_LTLSYN_OR1.ispl"  
    fo = open(filename, "w")  
    writeComment(fo,var1,var2,goal1,goal2)  
    createEnvironment(fo)  
    createPlayer(fo,var1,"1")  
    createPlayer(fo,var2,"2")  
    createEvaluation(fo,var1,var2)  
    createInitStates(fo,var1,var2)  
    createFormulae_LTLSYN_OR1(fo,goal1,goal2)  
    fo.close()
```

```
    # Run MCMAS
```

```
    done = runMCMAS(filename)  
    if done: return
```

```
    ### LTL SYN (OR2)
```

```
    filename = "input_LTLSYN_OR2.ispl"  
    fo = open(filename, "w")  
    writeComment(fo,var1,var2,goal1,goal2)  
    createEnvironment(fo)  
    createPlayer(fo,var1,"1")  
    createPlayer(fo,var2,"2")  
    createEvaluation(fo,var1,var2)  
    createInitStates(fo,var1,var2)  
    createFormulae_LTLSYN_OR2(fo,goal1,goal2)
```

```

fo.close()

# Run MCMAS
done = runMCMAS(filename)
if done: return

### LTL SYN (AND)
filename = "input_LTL SYN_AND.ispl"
fo = open(filename, "w")
writeComment(fo,var1,var2,goal1,goal2)
createEnvironment(fo)
createPlayer(fo,var1,"1")
createPlayer(fo,var2,"2")
createEvaluation(fo,var1,var2)
createInitStates(fo,var1,var2)
createFormulae_LTL SYN_AND(fo,goal1,goal2)
fo.close()

# Run MCMAS
done = runMCMAS(filename)
if done: return

# CTL* SYN (OR1)
filename = "input_CTL STAR_OR1.ispl"
fo = open(filename, "w")
writeComment(fo,var1,var2,goal1,goal2)
createEnvironment(fo)
createPlayer(fo,var1,"1")
createPlayer(fo,var2,"2")
createEvaluation(fo,var1,var2)
createInitStates(fo,var1,var2)
createFormulae_CTL Star SYN_OR1(fo,goal1,goal2)
fo.close()

# Run MCMAS
done = runMCMAS(filename)
if done: return

# CTL* SYN (OR2)
filename = "input_CTL STAR_OR2.ispl"
fo = open(filename, "w")
writeComment(fo,var1,var2,goal1,goal2)
createEnvironment(fo)
createPlayer(fo,var1,"1")
createPlayer(fo,var2,"2")
createEvaluation(fo,var1,var2)
createInitStates(fo,var1,var2)
createFormulae_CTL Star SYN_OR2(fo,goal1,goal2)

```

```
fo.close()

# Run MCMAS
done = runMCMAS(filename)
if done: return
```

A.2 main.py

```
### MAIN ###
from sourcecode import *

# Open user input file
print("_____")
print(str(time.time()) + ":parsing user input file...")
print("_____")

fo = open("input.ibg", "r")
content = fo.read().splitlines() # content in list

# Variables
singlerun = True

# Parse variables and goals
for line in content:
    if "Player1" in line:
        var1 = line.replace("Player1:", "").lstrip().split(";") #
            vars for player1
    elif "Player2" in line:
        var2 = line.replace("Player2:", "").lstrip().split(";") #
            vars for player2
    elif "Goal1" in line:
        goal1 = line.replace("Goal1:", "").lstrip() # goal for
            player1
    elif "Goal2" in line:
        goal2 = line.replace("Goal2:", "").lstrip() # goal for
            player2
    elif "Explore" in line:
        if "ALL" in line: singlerun = True
        elif "WHICH" in line: singlerun = False

# translate to .ispl file and run MCMAS
if singlerun:
    translateAll(var1,var2,goal1,goal2)
else:
    translateEach(var1,var2,goal1,goal2)
```

A.3 test.py

```
from sourcecode import *
import math

### Test Case 1 ###

# Simple goal(unchanged)
# Vary # of variables
# 1.number of variables growing linearly
# 2.number of variables growing exponentially

def testcase1():

    varlist1 = []
    varlist2 = []
    for i in range(100):
        varlist1.append("x" + str(i))
        varlist2.append("y" + str(i))
    goal1 = "x0"
    goal2 = "!y0"

    var1 = varlist1[0:7]
    var2 = varlist2[0:7]
    print var1
    print var2
    translateEach(var1, var2, goal1, goal2)

    # 1:n setting
    var1 = varlist1[0:6]
    for i in range(6):
        var2 = varlist2[0:(i+1)] # linear
        #var2 = varlist2[0:int(math.pow(2,i)))] # exponential
        print var1
        print var2
        translateEach(var1, var2, goal1, goal2)

    # n:n setting
    for i in range(10):
        # linear
        var1 = varlist1[0:(i+1)]
        var2 = varlist2[0:(i+1)]
        print var1
        print var2
        translateAll(var1, var2, goal1, goal2)

### Test Case 2 ###
```

```

# Simple varibales(unchanged)
# Vary the scope(complexity) of goals
# 1.Only with F
# 2.Only with G
# 3.Only with U
# 4.Full LTL: X,U,F,G

def testcase2_1():
    """
    var1 = ["x0","x1"]
    var2 = ["y0","y1"]
    goallist1 = ["F x0","F x0 and y1","F (x0 and y1) or !y0","F (
        y0 and x1) or F!y0 and x1"]
    goallist2 = ["F !y0","F !y0 and x1","F (!y0 and x0) or !x1","
        F (!y0 and x1) or F y1 and !x0"]
    """

    var1 = ["x0","x1","x2"]
    var2 = ["y0","y1","y2"]
    goallist1 = ["F x0 and y1","F (x0 and y1) or !y2","F (y1 and
        x1) or F !y2 and x2"]
    goallist2 = ["F !y0 and x1","F (!y0 and x1) or !x2","F (!y0
        and x0) or F y2 and !x1"]
    for i in range(len(goallist1)):
        goal1 = goallist1[i]
        goal2 = goallist2[i]
        print goal1
        print goal2
        translateAll(var1,var2,goal1,goal2)

def testcase2_2():
    """
    var1 = ["x0","x1"]
    var2 = ["y0","y1"]
    goallist1 = ["G x0","G x0 and y1","G (x0 and y1) or !y0","G (
        y0 and x1) or G!y0 and x1"]
    goallist2 = ["G !y0","G !y0 and x1","G (!y0 and x0) or !x1","
        G (!y0 and x1) or G y1 and !x0"]
    """
    """
    var1 = ["x0","x1","x2"]
    var2 = ["y0","y1","y2"]
    goallist1 = ["G x0 and y1","G (x0 and y1) or !y2","G (y1 and
        x1) or G !y2 and x2","G (y1 and x1) or G (!y2 and x2) and
        (x0 or y0)"]
    goallist2 = ["G !y0 and x1","G (!y0 and x1) or !x2","G (!y0
        and x0) or G (y2 and !x1)","G (!y0 and x0) or G (y2 and !

```

```

        x1) and (x2 or y1)"]
    ,,,
var1 = ["x0","x1","x2","x3"]
var2 = ["y0","y1","y2","y3"]
goallist1 = ["x0","G x0","G x0 and y1","G (x0 and y1) or !y2"
    ,"G (y1 and x1) or G !y2 and x2","G (y1 and x1) or G (!y2
    and x2) and (x0 or y0)"]
goallist2 = ["!y0","G !y0","G !y0 and x1","G (!y0 and x1) or
    !x2","G (!y0 and x0) or G (y2 and !x1)","G (!y0 and x0) or
    G (y2 and !x1) and (x2 or y1)"]

for i in range(len(goallist1)):
    goal1 = goallist1[i]
    goal2 = goallist2[i]
    print goal1
    print goal2
    translateAll(var1,var2,goal1,goal2)

def testcase2_3():
    ,,,
    var1 = ["x0","x1"]
    var2 = ["y0","y1"]
    goallist1 = ["x0 U y0","(x0 U y1) or (!y0 U x1)","((x0 U y0)
        and !y1 ) or (y1 U !x1)"]
    goallist2 = ["!x0 U !y0","(!x0 U !y1) or (y0 U x1)","((!x0 U
        !y0) and x1 ) or (!y1 U x1)"]
    ,,,

    var1 = ["x0","x1","x2"]
    var2 = ["y0","y1","y2"]
    goallist1 = ["x0 U y0","(x0 U y1) or (!y0 U x1)","((x0 U y0)
        and !y1 ) or (y2 U !x1 and x2)"]
    goallist2 = ["!x0 U !y0","(!x0 U !y1) or (y0 U x1)","((!x0 U
        !y0) and x1 ) or (!y1 U x2 and y2)"]

    for i in range(len(goallist1)):
        goal1 = goallist1[i]
        goal2 = goallist2[i]
        print goal1
        print goal2
        translateAll(var1,var2,goal1,goal2)

def testcase2_4():
    ,,,
    var1 = ["x0","x1"]
    var2 = ["y0","y1"]
    goallist1 = ["G F (x0 U y0)","G F (x0 U y1) and X(!y0 U x1)
        ","G F ((x1 U y0) and !y1 ) or ( X y1 U !x0)"]

```



```

goallist2 = ["G F (!x0 U !y0)", "G F (!x0 U !y1) or X(y1 U x1)",
            ", "G F ((!x1 U !y0) and x0 ) or X (!y1 U x0)"]
,,,

```

```

var1 = ["x0", "x1", "x2"]
var2 = ["y0", "y1", "y2"]
goallist1 = ["G F (x0 U y0)", "G F (x0 U y1) and X(!y0 U x1)",
            "G F ((x1 U y0) and !y1 ) or ( X y1 U !x0)"]
goallist2 = ["G F (!x0 U !y0)", "G F (!x0 U !y1) or X(y2 U x1)",
            ", "G F ((!x1 U !y0) and x0 ) or X (!y1 U x0)"]

```

```

for i in range(len(goallist1)):
    goal1 = goallist1[i]
    goal2 = goallist2[i]
    print goal1
    print goal2
    translateAll(var1, var2, goal1, goal2)

```

Test Case 3

Compare the performance between ALL and WHICH options

```

def testcase3():
    ,,,
    var1 = ["x0", "x1"]
    var2 = ["y0", "y1"]
    goallist1 = ["G F (x0 U y0)", "G F (x0 U y1) and X(!y0 U x1)",
                ", "G F ((x1 U y0) and !y1 ) or ( X y1 U !x0)"]
    goallist2 = ["G F (!x0 U !y0)", "G F (!x0 U !y1) or X(y1 U x1)",
                ", "G F ((!x1 U !y0) and x0 ) or X (!y1 U x0)"]
    ,,,

```

```

var1 = ["x0", "x1", "x2"]
var2 = ["y0", "y1", "y2"]
goallist1 = ["G F (x0 U x1)", "G F (x0 U y1) and X(!y0 U x1)",
            "G F ((x1 U y0) and !y1 ) or ( X y1 U !x0)"]
goallist2 = ["G F (!y0 U !y1)", "G F (!x0 U !y1) or X(y2 U x1)",
            ", "G F ((!x1 U !y0) and x0 ) or X (!y1 U x0)"]

```

```

for i in range(len(goallist1)):
    goal1 = goallist1[i]
    goal2 = goallist2[i]
    print goal1
    print goal2
    translateAll(var1, var2, goal1, goal2)

```

```
### Main ###  
testcase1()  
testcase2_1()  
testcase2_2()  
testcase2_3()  
testcase2_4()  
testcase3()
```

A.4 sample input.ibg

User Input File

Player1: x

Player2: y

Goal1: G F (x and y)

Goal2: G F (!x and !y)

Explore: ALL

A.5 sample input.ispl

```
-- Input iBG file
-- Player1: x
-- Player2: y
-- Goal1: G F (x and y)
-- Goal2: G F (!x and !y)

Agent Environment
  Obsvars:
    dummy : boolean;
  end Obsvars
  Actions = {none};
  Protocol:
    Other : {none};
  end Protocol
  Evolution:
    dummy = true if Action=none;
  end Evolution
end Agent
```

```
Agent Player1
  Vars:
    x : boolean;
  end Vars
  Actions = {ac0, ac1};
  Protocol:
    Other : {ac0, ac1};
  end Protocol
  Evolution:
    (x = false) if Action=ac0;
    (x = true) if Action=ac1;
  end Evolution
end Agent
```

```
Agent Player2
  Vars:
    y : boolean;
  end Vars
  Actions = {ac0, ac1};
  Protocol:
    Other : {ac0, ac1};
  end Protocol
  Evolution:
```

```

        (y = false) if Action=ac0;
        (y = true) if Action=ac1;
    end Evolution
end Agent

```

```

Evaluation
    x if Player1.x = true;
    y if Player2.y = true;
end Evaluation

```

```

InitStates
    (Environment.dummy = true) and
    ((Player1.x = true) or (Player1.x = false)) and
    ((Player2.y = true) or (Player2.y = false));
end InitStates

```

```

Formulae
    — LTL SAT
    <<strategy_env>> (Environment, strategy_env) (
        ( <<strategy_p1>> (Player1, strategy_p1) <<strategy_p2>> (
            Player2, strategy_p2) X ( (G F (x and y) ) and (G F (!x
            and !y)) ) )
    );

    — LTL SYN
    <<strategy_env>> (Environment, strategy_env) (
        ( <<strategy_p1>> (Player1, strategy_p1) [[strategy_p2]] (
            Player2, strategy_p2) X (G F (x and y) ) )
    );
    <<strategy_env>> (Environment, strategy_env) (
        ( <<strategy_p2>> (Player2, strategy_p2) [[strategy_p1]] (
            Player1, strategy_p1) X (G F (!x and !y)) )
    );

    — LTL SYN
    <<strategy_env>> (Environment, strategy_env) (
        ( <<strategy_p1>> (Player1, strategy_p1) [[strategy_p2]] (
            Player2, strategy_p2) X (!(G F (!x and !y)) ) )
    );
    <<strategy_env>> (Environment, strategy_env) (
        ( <<strategy_p2>> (Player2, strategy_p2) [[strategy_p1]] (
            Player1, strategy_p1) X (!(G F (x and y) ) ) )
    );

    — CTL* SYN

```

```

<<strategy_env>> (Environment, strategy_env) (
  ( <<strategy_p1>> (Player1, strategy_p1) ( [[strategy_p2]] (
    Player2, strategy_p2) X (!(G F (!x and !y)) ) and <<
    strategy_p2>> (Player2, strategy_p2) X (G F (x and y) ) )
  )
);
<<strategy_env>> (Environment, strategy_env) (
  ( <<strategy_p2>> (Player2, strategy_p2) ( [[strategy_p1]]
    (Player1, strategy_p1) X (!(G F (x and y) ) ) and <<
    strategy_p1>> (Player1, strategy_p1) X (G F (!x and !y)
    ) ) )
);

```

end Formulae

Appendix B

User Instructions

B.1 Set Up

1. Download MCMAS-SLK toolkit from: <http://vas.doc.ic.ac.uk/software/extensions/>. The installation guide can be found at: <http://vas.doc.ic.ac.uk/software/mcmas/documentation/>.
2. Download the source code from Github: <https://github.com/tonggao1024/Thesis>. Assume that the MCMAS package is unpacked to the directory ‘.../MCMAS’, the source code should be stored under directory ‘.../MCMAS/examples/iBG’.
3. Modify the user input file ‘.../MCMAS/examples/iBG/input.ibg’. For the variables, each variable is separated by ‘;’. For the ‘Explore’ feather, the keywords are [ALL] and [WHICH]. The user is allowed to choose either one of them.

B.2 Run MCMAS

First, jump to the following directory:

‘.../MCMAS/examples/iBG/’

Second, run the program with the following command:

‘python main.py’

B.3 Result Collection

The user can collect the results from the terminal window.

For keyword [ALL], the program will produces one of the following outputs:

‘The game has a Nash Equilibrium’

‘The game doesn’t have a Nash Equilibrium’

For keyword [WHICH], the program will produces one of the following outputs:

‘The game has a Nash Equilibrium, formula XXX is satisfied’

‘The game doesn’t have a Nash Equilibrium, formula XXX is satisfied’

Bibliography

- [1] L. Turocy Theodore and von Stengel Bernhard. Game theory, 2001.
- [2] Nash equilibrium. <http://www.investopedia.com/terms/n/nash-equilibrium.asp>. Accessed: 2016-08-25.
- [3] Dana Fisman, Orna Kupferman, and Yoad Lustig. Rational synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 190–204. Springer, 2010.
- [4] Julian Gutierrez, Paul Harrenstein, and Michael Wooldridge. Iterated boolean games. *Information and Computation*, 242:53–79, 2015.
- [5] Julian Gutierrez, Paul Harrenstein, and Michael Wooldridge. Expressiveness and complexity results for strategic reasoning. In *LIPICs-Leibniz International Proceedings in Informatics*, volume 42. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
- [6] Michael Wooldridge, Julian Gutierrez, Paul Harrenstein, Enrico Marchioni, Giuseppe Perelli, and Alexis Toumi. Rational verification: From model checking to equilibrium checking. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2016.
- [7] John F Nash et al. Equilibrium points in n-person games. *Proc. Nat. Acad. Sci. USA*, 36(1):48–49, 1950.
- [8] S.I. Briceno and Georgia Institute of Technology. *A Game-based Decision Support Methodology for Competitive Systems Design*. Georgia Institute of Technology, 2008. ISBN 9781109011418. URL <https://books.google.co.uk/books?id=y0jIuav484IC>.
- [9] Alvin E Roth and J Keith Murnighan. Equilibrium behavior and repeated play of the prisoner’s dilemma. *Journal of Mathematical psychology*, 17(2):189–198, 1978.
- [10] Amir Pnueli and Zohar Manna. The temporal logic of reactive and concurrent systems, 1992.
- [11] Ashari Rehab and Habib Sahar. Chapter 5 linear temporal logic (ltl).

- [12] Lecture 7: Computation tree logics.
- [13] Krishnendu Chatterjee, Thomas A Henzinger, and Nir Piterman. Strategy logic. *Information and Computation*, 208(6):677–693, 2010.
- [14] Jörg Flum, Erich Grädel, and Thomas Wilke. *Logic and automata: history and perspectives*. Amsterdam University Press, 2007.
- [15] Fabio Mogavero, Aniello Murano, and Moshe Y Vardi. Reasoning about strategies. In *LIPICs-Leibniz International Proceedings in Informatics*, volume 8. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.
- [16] Moshe Y Vardi. Automata-theoretic model checking revisited. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 137–150. Springer, 2007.
- [17] Igor Walukiewicz. A landscape with games in the background. In *Proceedings of LICS*, volume 4, pages 356–366, 2004.
- [18] Yves Deville and Kung-Kiu Lau. Logic program synthesis. *The Journal of Logic Programming*, 19:321–350, 1994.
- [19] Julian Gutierrez, Giuseppe Perelli, and Michael Wooldridge. Imperfect information in reactive modules games. 2016.
- [20] Alessio Lomuscio and Franco Raimondi. Mcmas: A model checker for multi-agent systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–454. Springer, 2006.
- [21] *MCMA5 v1.2.2: User Manual*. Imperial College London.
- [22] Petr Čermák, Alessio Lomuscio, Fabio Mogavero, and Aniello Murano. Mcmas-slκ: A model checker for the verification of strategy logic specifications. In *International Conference on Computer Aided Verification*, pages 525–532. Springer, 2014.
- [23] Gutierrez Julian, Harrenstein Paul, and Wooldridge Michael. From model checking to equilibrium checking: Verifying temporal equilibrium properties of multi-agent systems. 2016.
- [24] Chapter 4: Pure strategies and mixed strategy equilibrium.