Equilibrium Checking in Reactive Modules Games

Alexis TOUMI
Computer Science Department, University of Oxford

May 25, 2015

Abstract

In this project, we investigate *Reactive Modules Games* (RMGs), a framework for modelling concurrent multi-agent systems in which each agent has its own goals about the overall behaviour of the system, and thus behaves strategically in order to achieve this goal. Game theory provides the foundations for analysing the properties of such systems, and for defining what it means for the agents to behave *rationally*.

More particularly, we focus on the problem of *Equilibrium Checking*: checking whether a set of strategies forms a *Nash Equilibrium* for a given game. Intuitively, if the agents play according to a Nash equilibrium, then no player has any interest in changing its strategy. Thus, if we design a protocol which is a Nash equilibrium, we know that there is no rational reason for an agent to deviate from it.

We implemented a proof-of-concept tool for solving this problem. In the course of this implementation, we came up with a revised proof for the lemma which defines the semantics of RMGs in terms of a *Computation Tree Logic* theory — our main contribution to the theory of Reactive Modules Games.

I would like to honour the memory of John Forbes Nash, Jr., who died in a tragic car accident the night before completing this report. This project would not have existed without his pioneering work on equilibrium in game theory, and he will remain as a great source of inspiration.

Contents

Introduction

Invented at first for the mechanical manipulation of symbols, computers turned out to be wonderful tools for communication. This insight led to the development of the network of networks of computers which most of our communication today relies on — the Internet. To study the Internet assuming that every computer behaves honestly would be naive, but if we assume that every computer is dishonest then there is no point in communicating at all. A more reasonable assumption is that computers in the network are simply selfish: they behave in the way that serves best their own interests.

In this project, we investigate the properties of *computational economies*: multi-agent systems in which the agents (the players) have their own preferences (their own goals) about the overall behaviour of the system (the game). This playful metaphor is not fortuitous: indeed, the theory of games as first introduced by Morgenstern and von Neumann in their pioneering work *Theory of Games and Economic Behavior* [14] provides the theoretical concepts required to understand the properties of such systems.

First, let us define some of these concepts. A (theoretical) *game* can be defined by: *(i)* the set of *players*, *(ii)* the set of *moves* available to each players at any point of the game and *(iii)* the *outcome* of the game for each player, as a function of the players' moves. Given such a game, a *strategy* is a complete description of a player's choices, as a function of the other players' moves. Hence, once every player has chosen a strategy, the outcome of the game is completely determined.

At first a theory of human strategic behaviour — e.g. in military or economic wars — game theory turned out to share a deep interface with pure mathematics, and more recently, with theoretical computer science and artificial intelligence. One example in mathematics would be the work of Conway in *On Numbers and Games* [7], where he defines numbers (not only natural and real, but also "surreal" numbers) as a subclass of a more general concept: two-player games. As for theoretical computer science, the most striking example is *Game Semantics* (an area of ongoing research in this department) where game-theoretic concepts are being applied to the denotational semantics of programming languages. (The interface between game theory and artificial intelligence will be discussed in the *Related Work* section that follows.)

Our work is part of the RACE project (*Reasoning About Computational Economies*), a five-year ERC Advanced Grant aimed at developing the techniques required to understand the properties of game-like distributed systems, as well as developing a practical and robust system for the formal specification, verification, and analysis of such systems. One of the potential applications would be *Mechanism Design*: given the individual goals of the agents, we want to design a game in which the agents achieve some desired overall property, while individually pursuing their own best interest — see Wooldridge *et al.* [20].

In this project, we investigate a formalism introduced by Gutierrez *et al.* [10] to model these systems: *Reactive Modules Games* (RMGs). In an RMG, the players are finite automaton which assign values to Boolean variables for an infinite number of rounds. Their goal is expressed as a temporal logic formula: if the structure induced by the players' choices satisfies this formula, then the player is satisfied. Crucially, whether one player's goal is satisfied depends on the values of the variables controlled by the other players: this naturally leads to strategic behaviour.

This project focuses on the concept of *Nash Equilibrium* for reactive modules games. Intuitively, a set of strategies forms a Nash equilibrium for a given game if no player can benefit from changing its strategy, provided that the other players adhere to theirs. For instance, consider a standard protocol published by some known authority such as IEEE. When implementing this protocol, a programmer is likely to assume that his program will interact with other programs that implement the same standard. If this protocol forms a Nash equilibrium, then the programmer will have no rational incentive to write a program that deviates from the standard. (This example we borrow from [3].)

More particularly, this project investigates the problem of *Equilibrium Checking*: checking whether a set of strategies forms a Nash equilibrium for a given RMG.

Objectives

I To study the theory of reactive modules games as presented by Gutierrez *et al.* in [10]. In order to do this, we had to investigate some theoretical concepts that were outside our taught curriculum: the analysis of multi-agent systems, formal verification and model checking, branching-time temporal logics; and — perhaps most importantly — we had to familiarise ourselves with the main concepts of game theory.

II To investigate thoroughly the problem of equilibrium checking in reactive modules games, to study the algorithm used to solve this problem, and to give a detailed analysis of its computational complexity.

III To implement this algorithm, building a tool which would be a preliminary proof-of-concept for the RACE system. In the course of this implementation, we came up with a revised proof for one of the lemmas in [10] (Lemma 12 of this report). This is our main contribution to the theory of reactive modules games.

Overview

Background

We begin by introducing the main theoretical tools that we use throughout this project. First, we define the syntax and the semantics of propositional logic. Then, we introduce Kripke structures — labelled transition systems over which we interpret the *Computation Tree Logic* (CTL) formulae used to express the players' goals. Finally, we present two decision problems associated with CTL, *satisfiability* and *model checking*, and we discuss their time complexity.

Reactive Modules Games

We then introduce the language we use to specify game arenas — namely *Simple Reactive Modules Language* (SRML). We formally define the semantics of SRML arenas, and then give two equivalent semantics: one based on Kripke structures and another based on CTL. Finally, we introduce *Reactive Modules Games* (RMGs) and we give a formal definition of the notions of *strategies*, *goals* and *outcomes*.

Equilibrium Checking

We give a formal definition of the notion of *Nash equilibrium* (NE) in the context of reactive modules games, and we introduce the decision problem which is the main focus of this project, namely NE-MEMBERSHIP: checking whether a given set of strategies forms a Nash equilibrium for a given reactive modules game. We then present an algorithm for solving this problem and a proof that it belongs to the EXPTIME complexity class.

Case Study: A Peer-to-Peer Example

Finally, we present a simple example of a scenario that can be modelled by an RMG. We show three different sets of strategies which demonstrate the three possible cases in the equilibrium analysis of a game. From this analysis, we discuss the problem of symmetry in coordination games, and the possible refinements to the concept of Nash equilibrium.

Appendix

Appendix A presents the reasoning which led to our new proof of Lemma 12, and the consequences it has on the problem of equilibrium checking. In Appendix B, we describe the details of our implementation, and we discuss the results we obtained. Appendix C contains a copy of the Python code for our system.

Related Work

Our project draws on two closely related thread of computer science research: the first, coming from the artificial intelligence community, is the use of formal logic in the analysis of multi-agent systems; the second, coming from the formal verification community, is the use of game-theoretic ideas for the automated verification of computer programs.

Multi-Agent Systems. In his seminal article *Computing Machinery and Intelligence* [19] Turing gave the first definition of what artificial intelligence should be about: creating a software agent that seems intelligent to a human observer. However, a significant part of AI research now focuses on *multi-agent systems* and game theory allows us to define what it means for these agents to behave *rationally*.

Formal logic languages provide the tools for reasoning about such rational agents: their beliefs about how the world is and their desires about how it should be. In their survey *Towards a Logic of Rational Agency* [12], van der Hoek and Wooldridge present the state of the art in developing logical theories to design and to analyse rational agents.

"But rationality is just another word for artificial intelligence. [...] Therefore, a multi-agent system in which interactions are governed by game theory would be perceived as intelligent by a human observing it." [16]

Formal Verification. Formal verification focuses on the problem of checking whether a system *S* satisfies a given property ψ, where typically ψ is a temporal logic formula. These logics describe *temporal ordering of events*, e.g. "event *a* always follows event *b*". However, in the case of systems of rational agents, temporal logic formulae cannot express *strategic* properties such as "agent *i* can guarantee that event *a* never happens".

This motivated Alur *et al.* [2] to develop *Alternating-time Temporal logic* (ATL) in order to express such strategic properties explicitly. ATL formulae are interpreted over *concurrent game structures*, Kripke structures where each transition is associated with a set of choices, one for each player. However, ATL provides no direct mechanism for referring explicitly to strategies. Hence, it provides no natural way to express game-theoretic properties such as Nash equilibrium.

These concerns motivated the development of *Strategy Logic* [3], a temporal logic that treats strategies of two-player games as explicit first-order objects.

Iterated Boolean Games. Whereas both ATL and Strategy Logic formulae are interpreted over game structures explicitly modelled as graphs (which will be exponential in the number of variables in the system), in *Iterated Boolean Games* [11] (iBGs) the structure of the game is defined succinctly, simply by listing the variables each player controls.

This is the formalism most closely related to Reactive Modules Games. However, in iBGs players are assumed to have complete freedom over the values they assign to their controlled variables. This makes iBGs a very high-level abstraction of real multi-agent systems, and it limits the scenarios that can be modelled (as shown in the case study we present in Section 4).

The approach taken by Gutierrez *et al.* in [10], and that we follow in this project, is to model games succinctly using a guarded command language that is much closer to real-world programming and system modelling, namely *Simple Reactive Modules Language*.

1 Background

In this section, we present the theoretical tools used in this project: firstly, *Propositional Logic*; secondly, *Computation Tree Logic* (CTL) and its semantic basis on Kripke structures; finally, two decision problems concerning CTL: *satisfiability* and *model checking*.

1.1 Propositional Logic

We define propositional logic with respect to a finite set of Boolean variables Φ and we interpret propositional formulae with respect to a set $v \subseteq \Phi$, called a *valuation*. We say that variable $p \in \Phi$ is true under valuation v if $p \in v$, while $p \notin v$ means that p is false under v.

Let $v_\bot = \emptyset$ be the valuation under which all variables are false, and let $V(\Phi) = 2^\Phi$ be the set of all valuations for variables Φ; where Φ is clear from the context we omit it and write V.

Definition 1 *The syntax of propositional logic formulae is defined as follows:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \qquad \textit{(where } p \in \Phi\textit{)}$$

We denote the set of propositional logic formulae over Φ by $\textsc{Pl}(\Phi)$; where Φ is clear from the context we omit it and write PL.

We will write "$\top$" for the formula $(p \vee \neg p)$, and "$\bot$" for the formula $(\neg\top)$. We also define the connectives "$\wedge$", "$\rightarrow$" and "$\leftrightarrow$" in the usual way:

$$\begin{array}{lcl} \varphi \wedge \psi & := & \neg(\neg\varphi \vee \neg\psi) \\ \varphi \rightarrow \psi & := & \neg\varphi \vee \psi \\ \varphi \leftrightarrow \psi & := & (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) \end{array}$$

Definition 2 *We write $v \models \varphi$ to mean that $\varphi \in$ PL is satisfied by a valuation $v \in V$:*

$$\begin{array}{lll} v \models p & \textit{iff} & p \in v \qquad \textit{(where } p \in \Phi\textit{)} \\ v \models \neg\varphi & \textit{iff} & \textit{it is not the case that } v \models \varphi \\ v \models \varphi \vee \psi & \textit{iff} & v \models \varphi \quad \textit{or} \quad v \models \psi \end{array}$$

1.2 Kripke Structures

Definition 3 *A Kripke Structure over Φ is a 4-tuple (S, S^0, R, π) where*

$$\begin{array}{ll} S \neq \emptyset & \textit{is a finite set of states} \\ S^0 \subseteq S & \textit{is the set of initial states} \\ \pi : S \rightarrow V & \textit{is the valuation function} \\ R \subseteq S \times S & \textit{is a total transition relation} \\ & \textit{(i.e. } \forall x \in S\ \exists y \in S \cdot x\ R\ y\textit{)} \end{array}$$

A *run* of $K = (S, S^0, R, \pi)$ is an infinite sequence of states $\rho = \rho_0, \rho_1, \rho_2, ...$ such that for all $t \in \mathbb{N}$ we have $\rho_t \in S$ and $(\rho_t, \rho_{t+1}) \in R$. We say ρ is an s-run if $\rho_0 = s$, and if $\rho_0 \in S^0$ we say ρ is an initial run. Let $runs(K, s)$ be the set of s-runs of K, and let $runs(K)$ be the set of initial runs of K. Each run $\rho \in runs(K)$ induces an infinite sequence of valuations $\boldsymbol{\rho} = \boldsymbol{\rho_0}, \boldsymbol{\rho_1}, \boldsymbol{\rho_2}, ...$ where for all $t \in \mathbb{N}$, $\boldsymbol{\rho}_t = \pi(\rho_t)$. We denote the set of these sequences by $\mathbf{runs}(K)$, using bold font to denote the difference with $runs(K)$.

Here, we define a *tree* as a non-empty set of integer strings $T \subseteq \mathbb{N}^*$ where *(i)* T is closed under prefixes, i.e. for every $u, v \in \mathbb{N}^*$, $uv \in T \rightarrow u \in T$, *(ii)* T is infinite, i.e. for every $u \in T$ there is an $x \in \mathbb{N}$ such that $ux \in T$. Note that T only defines the set of vertices, edges are only given implicitly: there is an edge between $u, v \in T$ whenever there is an $x \in \mathbb{N}$ with $ux = v$.

Definition 4 *A state-tree for a Kripke structure (S, S^0, R, π) is a function $\kappa : T \rightarrow S$ where*

(i) $T \subseteq \mathbb{N}^*$ is a tree
(ii) $\kappa(\epsilon) \in S^0$
(iii) $\forall ux \in \mathbb{N}^* \cdot ux \in T \rightarrow \kappa(u)\ R\ \kappa(ux)$
(iv) $\forall ux, uy \in \mathbb{N}^* \cdot \kappa(uy) = \kappa(ux) \rightarrow x = y$

We denote the set of state-trees for K by $trees(K)$.

Every state-tree $\kappa : T \rightarrow S$ induces a *computation tree* $\boldsymbol{\kappa} : T \rightarrow V(\Phi)$ such that, for every $u \in T$ we have $\boldsymbol{\kappa}(u) = \pi(\kappa(u))$. We denote the set of computation trees for K by $\mathbf{trees}(K)$. We define an *unfolding* of $K = (S, S^0, R, \pi)$ to be a maximal computation tree:

1) Let $\kappa : T \rightarrow S$ be a maximal state-tree for K, i.e. for every $u \in T$ and every $s \in S$, $\kappa(u)\ R\ s$ implies that there is some $x \in \mathbb{N}$ with $ux \in T$ and $\kappa(ux) = s$.

2) The computation tree $\boldsymbol{\kappa} : T \rightarrow V$ induced by κ is an unfolding of K. Note that every initial state $s \in S^0$ induces a unique unfolding.

We denote the set of unfoldings for K by $\mathbf{unfold}(K)$. We also define the size of a Kripke structure $K = (S, S^0, R, \pi)$, written $|K|$, to be $|S| + |R|$.

1.3 Computation Tree Logic

Computation Tree Logic (CTL) is a branching-time logic which models time as a tree-like structure in which the future is non-deterministic. CTL formulae essentially express predicates over computation trees by extending propositional logic with two *path quantifiers*: $\mathbf{A}$ ("on all paths...") and $\mathbf{E}$ ("on some path..."), and two *modal tense operators* : $\mathbf{X}$ ("next") and $\mathbf{U}$ ("until").

Definition 5 *The syntax of CTL formulae is given by the following grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{AX}\varphi \mid \mathbf{E}(\varphi\, \mathbf{U}\, \varphi) \mid \mathbf{A}(\varphi\, \mathbf{U}\, \varphi) \qquad \textit{(where } p \in \Phi\textit{)}$$

We denote the set of CTL formulae over Φ by $\mathrm{CTL}(\Phi)$; where Φ is clear from the context we omit it and write CTL.

Remark 6 *Notice that the syntax of CTL formulae enforces that path quantifiers strictly alternate with modal operators. We can derive the pair "***EX***" from De Morgan's laws:* $\mathbf{EX}\psi = \neg\mathbf{AX}\neg\psi$. *We also derive* **G** *("always") and* **F** *("eventually"), as follows:*

$$\begin{aligned}\mathbf{AF}\psi &= \mathbf{A}(\top\,\mathbf{U}\,\psi)\\ \mathbf{EF}\psi &= \mathbf{E}(\top\,\mathbf{U}\,\psi)\\ \mathbf{AG}\psi &= \neg\mathbf{EF}\neg\psi\\ \mathbf{EG}\psi &= \neg\mathbf{AF}\neg\psi\end{aligned}$$

We interpret formulae with respect to a pair (K, s) where K is a Kripke structure over Φ and s is a state of K. We write $(K, s) \models \psi$ to mean that a formula ψ is *satisfied* in state s of K:

$(K,s) \models p$	iff	$p \in \pi(s)$ (where $p \in \Phi$)
$(K,s) \models \neg\psi$	iff	it is not the case that $(K,s) \models \psi$
$(K,s) \models \psi \vee \varphi$	iff	$(K,s) \models \psi$ or $(K,s) \models \varphi$
$(K,s) \models \mathbf{AX}\psi$	iff	for all $\rho \in \mathit{runs}(K,s)$ we have $(K,\rho_1) \models \psi$
$(K,s) \models \mathbf{A}(\psi\,\mathbf{U}\,\varphi)$	iff	for all $\rho \in \mathit{runs}(K,s)$ there is some $u \in \mathbb{N}$ with $(K,\rho_u) \models \varphi$ and for all $0 \le v < u \cdot (K,\rho_v) \models \psi$
$(K,s) \models \mathbf{E}(\psi\,\mathbf{U}\,\varphi)$	iff	there is some $\rho \in \mathit{runs}(K,s)$ and $u \in \mathbb{N}$ with $(K,\rho_u) \models \varphi$ and for all $0 \le v < u \cdot (K,\rho_v) \models \psi$

If $(K, s_0) \models \psi$ for all $s_0 \in S^0$, we say that K *satisfies* ψ, written $K \models \psi$. We say a formula $\psi \in$ CTL is *satisfiable* if $K \models \psi$ for some K. We define the *size of a formula* $\psi \in$ CTL, written $|\psi|$, to be the number of its subformulae.

1.4 SAT Solving

A natural decision problem associated with CTL is the satisfiability problem: given a CTL formula γ, is there a Kripke structure K that satisfies it?

CTL SAT
Given: A CTL formula γ
Question: Is formula γ satisfiable?

The equilibrium checking algorithm presented in Section 3.2 assumes an oracle for CTL SAT, and our implementation will treat the algorithm for solving it as a black box. We will need the following complexity result:

Theorem 7 (From [8]) CTL SAT $\in$ EXPTIME.

Proof: We will not give a detailed proof here, and will only present a brief outline of the algorithm presented in [8]. A way to define an EXPTIME algorithm for CTL SAT follows the *tableau method*: it tries to build a satisfying Kripke structure where each state is labelled with the set of formulae that must hold in that state.

Starting with a single state labelled with the given CTL formula, it then expands the tableau by applying a finite set of rules to break formulae into their subformulae, creating new states when necessary, until it reaches a contradiction or until no more rules apply, in which case the Kripke structure thus generated satisfies the given formula.

That the algorithm runs in EXPTIME follows from the fact that we need to apply a number of rules at most exponential in the size of the formula. ∎

1.5 Model Checking

The equilibrium checking algorithm of Section 3.2 also assumes an oracle for another decision problem associated with CTL, namely *model checking*:

CTL MC
Given: A Kripke structure K and CTL formula γ
Question: Is it the case that $K \models \gamma$?

We will also treat the algorithm for solving CTL MC as a black box, and only give an outline of the proof of the following complexity result:

Theorem 8 (From [8]) CTL MC $\in$ PTIME.

Proof: A polynomial-time algorithm for checking if $(K, \gamma) \in$ CTL MC works by labelling every state of K with the subformulae of γ that hold in that state, working in a bottom-up way on the syntax tree of γ.

Suppose the states satisfying all the subformulae of γ have already been labelled. We can determine which states satisfy the formula γ using a specific rule for each case in the grammar of CTL. Then if all the initial states of K are labelled with γ, we have that $K \models \gamma$.

The fact that the algorithm works in polynomial time follows from the fact that applying one of these rules is at most quadratic in the size of the Kripke structure, and that we apply exactly one rule per subformula. ∎

2 Reactive Modules Games

In this section, we describe the language we use to specify players and their choices: *Simple Reactive Modules Language*. Our formalisation of SRML will closely follow that in [10], however we will consider only the branching-time approach, and therefore we will not cover games with *Linear Temporal Logic* (LTL) goals. Our presentation of the material will focus on the technical results that are used in this project.

The games we investigate are played by a set $N = \{1, 2, ..., n\}$ of players, over a set Φ of Boolean variables, with each player $i \in N$ controlling a subset $\Phi_i \subseteq \Phi$. We require that $\{\Phi_1, ..., \Phi_n\}$ forms a partition of Φ, i.e. every variable is controlled by some player and no variable is controlled by more than one player.

The games are played in rounds, and we assume an infinite number of rounds. Prior to the game, each player selects a strategy that will specify how it makes choices over time, i.e. how to assign truth values to the variables it controls. The strategies we consider here are non-deterministic, so the playing of a game can be represented as a Kripke structure over Φ. The outcome for a player depends on whether its goal, expressed as a CTL formula, is satisfied by this induced structure.

Note that players' goals need not be contradictory: two formulae can be satisfied by the same Kripke structure, so two players can get their goal achieved simultaneously. Note also that whether a player gets its goal achieved crucially depends on the value of the variables controlled by other players: this naturally leads to strategic behaviour.

2.1 Guarded Commands and Reactive Modules

To specify the rules of the game, we specify for each player the set of choices available at each turn, which will depend on the values of the variables in Φ at this point of time. We define these choices using *guarded commands*:

Definition 9 *A guarded command g over Φ is an expression of the form:*

$$\varphi \rightsquigarrow x'_1 := \psi_1, ..., x'_k := \psi_k$$

where $\varphi \in \text{PL}(\Phi)$ and for all $i \leq k$, we have $x_i \in \Phi$ and $\psi_i \in \text{PL}(\Phi)$.

We say that the first component of a command g is the *guard*, denoted by $guard(g)$. The second component is a sequence of assignment statements called the *action* of the command. We require that an action never assigns the same variable more than once, and say that $\{x_1, ..., x_k\}$ are the *controlled variables of g*, denoted by $ctr(g)$.

When the guard of a command is satisfied by the current valuation of Φ, we say that the command is *enabled*, i.e. it may be chosen to be executed. Executing an enabled command means evaluating each of the ψ_i against the current state of the system, then in the next state we assign each variable x_i to the corresponding truth value thus obtained. (Here the "prime" notation x' means "the value of variable x *after* the command is executed.")

In SRML we use *modules* to define players. They consist of: *(i)* an interface which defines the name of the module and the subset of variables it controls, *(ii)* a list of ***init*** *guarded commands* used for initialising the variables under the module's control, and *(iii)* a list of ***update*** *guarded commands* for updating these variables at each round.

Whereas the full RML language used in model checkers such as MOCHA [1] and PRISM [13] provides complex initialisation schemes, in SRML we always interpret **init** commands with respect to the empty valuation $v_\perp$. Hence, we will assume that **init** commands are always enabled, i.e. the guard of an **init** command will always be the constant $\top$; we also require that, in every assignment $x := \psi$, the formula ψ is simply a Boolean constant, $\top$ or $\perp$. We introduce the concrete syntax for modules along with an example:

$$
\begin{array}{l}
\textbf{module } \textit{toggle} \textbf{ controls } x, y \\
\quad \textbf{init} \\
\quad :: \top \rightsquigarrow x' := \top,\ y' := \perp \\
\quad :: \top \rightsquigarrow x' := \perp,\ y' := \top \\
\quad \textbf{update} \\
\quad :: (x \wedge \neg y) \rightsquigarrow x' := \perp,\ y' := \top \\
\quad :: (\neg x \wedge y) \rightsquigarrow x' := y,\ y' := x
\end{array}
$$

This module, named *toggle*, controls two variables x and y. It has two **init** guarded commands and two **update** guarded commands. The **init** commands define two choices for the initialisation of the pair (x, y): assign it the value $(\top, \perp)$ or the value $(\perp, \top)$. The first **update** command says that if (x, y) has the value $(\top, \perp)$ then the corresponding choice is to assign it the value $(\perp, \top)$, while the second command says that if the pair (x, y) has the value $(\perp, \top)$, we can assign it the value (y, x) in the next state.

Note that the two **update** commands define essentially the same choice, but in the first command the action mentions Boolean constants directly, whereas the second command mentions the values of the variables at the current state, and requires to evaluate those to assign the values for the next state. In other words, the module *toggle* first nondeterministically picks an initial pair in $\{(\top, \perp), (\perp, \top)\}$, then at each round it deterministically toggles between these two pairs.

2.2 Formal Definition of SRML

Let us now define SRML modules formally. We first define an SRML *arena* as a collection of modules, one for each player $i \in N$, along with the set of variables they control:

Definition 10 *An arena A is given by an $(n+2)$-tuple $(N, \Phi, m_1, ..., m_n)$ where*

$N = \{1, 2, ..., n\}$ *is the set of players,*
Φ *is a finite set of propositional variables,*
and for each $i \in N$, m_i is a module, defined as follows:

A module m_i is defined as a triple (Φ_i, I_i, U_i) where

$\Phi_i \subseteq \Phi$ *is the set of variables controlled by m_i.*
I_i *is the set of* ***init*** *guarded commands.*
U_i *is the setmodel checking of* ***update*** *guarded commands.*

Such that for all $g \in I_i \cup U_i$, we have $ctr(g) \subseteq \Phi_i$.

We define the *size* of a module $m_i = (\Phi_i, I_i, U_i)$, written $|m_i|$, to be the sum of the sizes of its guarded commands, bounded by $\Phi_i * (I_i + U_i)$. We define the size of an arena to be the sum of the sizes of its constituent modules.

Given a module $m_i = (\Phi_i, I_i, U_i)$ with $\Phi_i = \{x_1, ..., x_k\}$, a set $C_i \subseteq I_i \cup U_i$ and a valuation $v \in V(\Phi)$, we define $enabled(C_i, v)$ as the set of guarded commands in C_i that are enabled at v. We require this set never to be empty, so we introduce a *null-guarded command* which leaves all the values for $x_1, ..., x_k \in \Phi_i$ unchanged. Formally, we have :

$$enabled(C_i, v) = \begin{cases} \{g \in C_i : v \models guard(g)\} & \text{if } \{g \in C_i : v \models guard(g)\} \neq \emptyset \\ \{\top \rightsquigarrow x_1' := x_1, ..., x_k' := x_k\} & \text{otherwise} \end{cases}$$

We now define a function that specifies the semantics of a guarded command. Given $g : \varphi \rightsquigarrow x_1' := \psi_1, ..., x_k' := \psi_k$, a guarded command in a module m_i that controls variables in Φ_i, we define $exec_i(g, v)$ to be the valuation obtained by executing command g on valuation v. The function $exec_i$ only returns the valuation for the variables Φ_i controlled by m_i, it does not specify the value of variables controlled by other modules. Formally:

$$exec_i(\varphi \rightsquigarrow x_1' := \psi_1, ..., x_k' := \psi_k) = \\ ((v \cap \Phi_i) \backslash ctr(g)) \cup \{x_i \in ctr(g) : v \models \psi_i\}$$

At each round, the behaviour of the arena is obtained by executing one command per module, in a concurrent and synchronous way. A *joint guarded command* is a profile $J = (g_1, ..., g_n)$ of guarded commands, one for each module in the arena. We say that a joint guarded command is enabled whenever each of its constituent commands is enabled. We extend the definition of *exec* in the obvious way:

$$exec(J, v) = \bigcup_{g_i \in J} exec_i(g_i, v)$$

We then execute the arena in a non-deterministic manner, producing a computation tree $\boldsymbol{\kappa} : T \to V$ such that *(i)* there is some enabled J with $exec(J, v_\perp) = \boldsymbol{\kappa}(\varepsilon)$ and *(ii)* for all $u, ux \in T$ there is some enabled J with $exec(J, \boldsymbol{\kappa}(u)) = \boldsymbol{\kappa}(ux)$. An arena A may allow multiple computation trees, the set of which we denote by $\mathbf{trees}(A)$.

2.3 Kripke-based Semantics

One of the reasons for using SRML is that it allows for the *succinct* description of a distributed, concurrent, and multi-agent system. However, when working with SRML arenas it is sometimes useful to refer explicitly to the Kripke structure that they induce.

We now present a lemma that describes how to get from an SRML arena to a Kripke structure, while preserving their induced computation trees.

Lemma 11 (From [10]) *For every* SRML *arena A, there is a Kripke structure K_A of size at most exponential in $|A|$ such that*

$$\mathbf{trees}(A) = \mathbf{trees}(K_A)$$

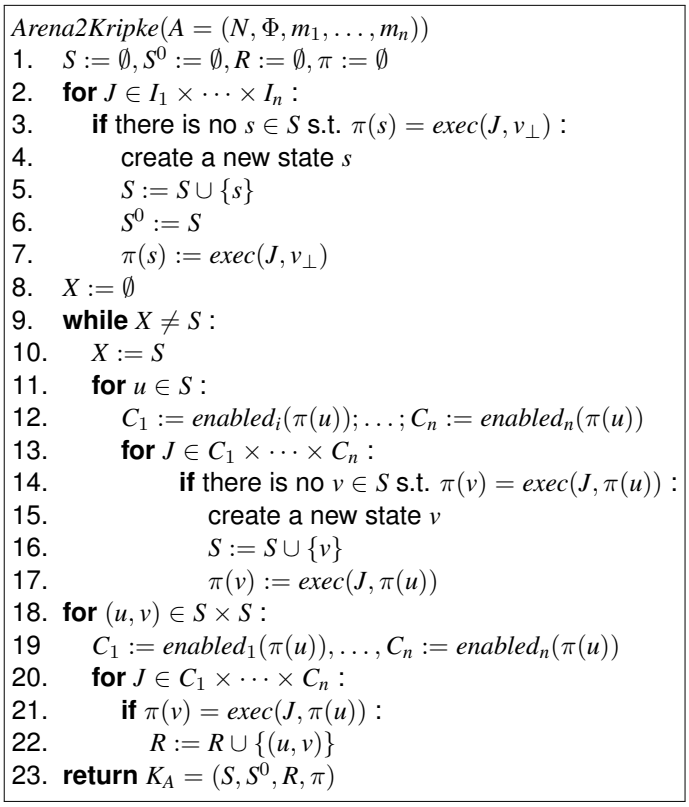

Figure 1: Algorithm for generating Kripke structures induced by an SRML arena.

Proof: Based on the formalisation of SRML, Figure 1 presents an exponential algorithm to construct, given an arena $A = (N, \Phi, m_1, ..., m_n)$, its induced Kripke structure K_A.

The states of the Kripke structure correspond to valuations of the variables in Φ, and the transition relation models the execution of guarded commands: for every states $u, v \in S$ we have $(u, v) \in R$ whenever there is a joint guarded command J, enabled at $\pi(u)$, such that $\pi(v) = exec(J, \pi(u))$.

The algorithm works by first creating the initial states of the structure (lines 2-7), one for every joint guarded command in the Cartesian product of the **init** commands of the modules. Then (lines 8-17), for every state u it goes through every joint guarded command J enabled at $\pi(u)$ and it creates a new state v with $\pi(v) = exec(J, u)$, until no new state is created. Finally (lines 18-22), it builds the transition relation R by going through all pairs (u, v) of states, adding the pair to R whenever there is a joint guarded command J enabled at $\pi(u)$ with $\pi(v) = exec(J, u)$.

To see that the induced Kripke structure K_A may be exponential in the size of the arena A, we observe that the set of states S cannot be bigger than the powerset of the controlled variables Φ, hence at most exponential in $|A|$. ∎

2.4 Logic-based Semantics

We now present another lemma that relates further the behaviour and the size of SRML arenas to that of their induced Kripke structure. We show how, given an arena A, we can compute a CTL formula $\text{TH}(A)$ that acts as the *theory* of A: the Kripke structures that satisfy $\text{TH}(A)$ are exactly those whose induced computation trees are in $\mathbf{trees}(A)$.

This lemma appears in [10], however the following proof fixes the problems that the original one encountered, as will be discussed in Appendix A. This is our main contribution to the theory of Reactive Module Games.

Lemma 12 *For every arena A, there is a CTL formula* $\text{TH}(A)$*, of size at most polynomial in the size of A such that*

$$\boldsymbol{\kappa} \in \mathbf{trees}(A) \iff K \models \text{TH}(A)$$

for every Kripke structure K and computation tree $\boldsymbol{\kappa} \in \mathbf{unfold}(K)$.

Proof: Let $A = (N, \Phi, m_1, ..., m_n)$ be the given arena. We will define $\text{TH}(A)$ as the conjunction of two predicates, which correspond to the semantics of the **init** and the **update** guarded commands:

$$\text{TH}(A) = \text{INIT}(A) \wedge \text{UPDATE}(A)$$

We begin with the semantics of the initialisation of the modules, given by $\text{INIT}(A)$. First, we define the effect of a single **init** command $g = \top \rightsquigarrow x'_1 := b_1, ..., x'_k := b_k$ for a module m_i:

$$\text{INIT}_i(g) = \bigwedge_{1 \le l \le k} (x_l \leftrightarrow b_l) \wedge \bigwedge_{x \in \Phi_i \setminus ctr(g)} (x \leftrightarrow \bot)$$

Then the predicate INIT_i asserts that exactly one **init** command is executed for a module $m_i = (\Phi_i, I_i, U_i)$:

$$\text{INIT}_i = \bigoplus_{g \in I_i} \text{INIT}_i(g)$$

Finally, we can define the initialisation predicate for the arena A:

$$\text{INIT}(A) = \bigwedge_{i \in N} \text{INIT}_i$$

We then define the semantics for the **update** commands, given by $\text{UPDATE}(A)$. We begin by defining the predicate $\text{ASSIGN}(x' := \psi)$, which asserts that, in the next round, the variable x is assigned the current value of ψ:

$$\text{ASSIGN}(x' := \psi) = (\psi \rightarrow \mathbf{AX}\, x) \wedge (\neg\psi \rightarrow \mathbf{AX}\neg x) \quad .$$

Hence, if the current valuation satisfies ψ then in the next round ($\mathbf{AX}$: "on all paths, next") x has the value $\top$, and if ψ is not satisfied then the next value of x is $\bot$. We also define the predicate $\text{UNCH}(\Psi)$, which asserts that the values of the variables in $\Psi \subseteq \Phi$ will remain unchanged in the next round:

$$\text{UNCH}(\Psi) = \bigwedge_{x \in \Psi} \text{ASSIGN}(x' := x) \quad .$$

For each **update** command $g = \varphi \rightsquigarrow x'_1 := \psi_1, ..., x'_k := \psi_k$ for module m_i, we define:

$$\textsc{Update}_i(g) = \varphi \wedge \left(\bigwedge_{1 \leq l \leq k} \textsc{Assign}(x'_l := \psi_l) \right) \wedge \textsc{Unch}(\Phi_i \backslash ctr(g)) \quad .$$

Hence, for a command $g \in U_i$ the formula $\textsc{Update}_i(g)$ asserts that: *(i)* the current valuation satisfies $guard(g)$, i.e. the command is enabled, *(ii)* the command is executed to obtain the values of the variables in $ctr(g)$ for the next round, and *(iii)* the variables controlled by the module but not mentioned by the command remain unchanged.

Then, $\textsc{Update}_i$ asserts that *(i)* if no **update** commands are enabled, then module m_i executes the null-guarded command, and *(ii)* otherwise it executes exactly one enabled **update** command:

$$\textsc{Update}_i = \left(\textsc{Unch}(\Phi_i) \wedge \bigwedge_{g \in U_i} \neg guard(g) \right) \vee \bigoplus_{g \in U_i} \textsc{Update}_i(g) \quad .$$

Finally, we can define the update predicate for the arena A:

$$\textsc{Update}(A) = \mathbf{AG} \bigwedge_{i \in N} \textsc{Update}_i \quad .$$

which asserts that at any point in time ($\mathbf{AG}$: "on all paths, always") the conjunction of the $\textsc{Update}_i$ predicates is satisfied, i.e. exactly one joint guarded command is executed to get to the next round.

The fact that $\textsc{Th}(A)$ is polynomial in $|A|$ immediately follows from the construction. To prove that for every K whose unfolding is $\boldsymbol{\kappa}$, $K \models \textsc{Th}(A)$ if and only if $\boldsymbol{\kappa} \in \mathbf{trees}(A)$, one proceeds by induction. ∎

Interestingly, our construction (as opposed to that of [10]) makes use only of the universal quantifier, hence it belongs to the common fragment of LTL and CTL. See Appendix A for a more thorough discussion of the implications this has for our project.

2.5 Goals, Strategies and Outcomes

So far, we have defined players and their choices and we have formalised SRML, the language for specifying them, but we have not defined the preferences of the players.

In a *Reactive Module Games* (RMG), we associate every player i with a *goal*: a CTL formula γ_i. We assume that a player will prefer an outcome that satisfies its goal over one that does not, and will be indifferent otherwise.

Definition 13 *An RMG G is a pair $(A, \vec{\gamma})$ where*

$$\begin{array}{ll} A = (N, \Phi, m_1, ..., m_n) & \textit{is an SRML arena.} \\ \vec{\gamma} = (\gamma_1, ..., \gamma_n) & \textit{is a profile of goals.} \end{array}$$

The game is played by each player i selecting a *strategy* σ_i that defines how to make choices over time. Once every player i has selected a strategy σ_i, we obtain a *strategy profile* $\vec{\sigma}$ and an *outcome* $K_{\vec{\sigma}}$ results.

Let us formalise this. We will write Φ_{-i} for $\Phi \backslash \Phi_i$ and let V_i, V_{-i} be the set of valuations for variables in Φ_i, Φ_{-i} respectively.

Definition 14 *A strategy σ_i for a module $m_i = (\Phi_i, I_i, U_i)$ is a 4-tuple $(Q_i, Q_i^0, \delta_i, \tau_i)$ where*

$Q_i \neq \emptyset$	*is a finite set of states.*
$Q_i^0 \subseteq Q_i$	*is the set of initial states.*
$\delta_i : Q_i \times V_{-i} \to \mathcal{P}(Q_i) \backslash \emptyset$	*is the (total) transition relation.*
$\tau_i : Q_i \to V_i$	*is the output function.*

Remark 15 *Note that a strategy for a given module will not necessarily satisfy that module's specification. For example, if the only guarded command of a module has the form $\top \rightsquigarrow x' := \top$, then a strategy for this module should never assign x the value $\bot$.*

Thus, given a module $m_i = (\Phi_i, I_i, U_i)$, a strategy $\sigma_i = (Q_i, Q_i^0, \delta_i, \tau_i)$ is *consistent with* m_i if the following conditions are satisfied: *(i)* for all initial states $q_i^0 \in Q_i^0$ there is a $g \in enabled(I_i, v_\bot)$ such that $exec(g, v_\bot) = \tau_i(q_i^0)$ and *(ii)* for all $q, q' \in Q_i$ and $v_i \in V_i$, $v_{-i} \in V_{-i}$ such that $\tau_i(q) = v_i$ and $q' \in \delta_i(q, v_{-i})$, there is some $g \in enabled(U_i, v_i \cup v_{-i})$ with $exec(g, v_i \cup v_{-i}) = \tau_i(q')$.

If a strategy σ_i is consistent with module m_i, we simply say that σ_i is a *strategy for* m_i. Given an arena $A = (N, \Phi, m_1, ..., m_n)$, we say that a strategy profile $\vec{\sigma} = (\sigma_1, ..., \sigma_n)$ is *consistent with A* if each σ_i is consistent with m_i. In this project, we will only consider consistent strategies.

We now define how a strategy profile $\vec{\sigma}$ extends straightforwardly to an outcome $K_{\vec{\sigma}}$: a Kripke structure that composes together the transition systems of the players' strategies.

Definition 16 *The outcome of a strategy profile $\vec{\sigma}$ is a structure $K_{\vec{\sigma}} = (S, S^0, \delta, \pi)$ where*

(i) $S = Q_1 \times ... \times Q_n$
(ii) $S^0 = Q_1^0 \times ... \times Q_n^0$
and for all states $q = (q_1, ..., q_n) \in S$, *we have:*
(iii) $\pi(q) = \bigcup_{i \in N} \tau_i(q_i)$
(iv) $\delta(q) = \prod_{i \in N} \delta_i(q_i, \pi(q))$

We say that a player *i* gets its goal γ_i satisfied by a strategy profile $\vec{\sigma}$ if the outcome $K_{\vec{\sigma}}$ satisfies the CTL formula γ_i. We will write $\vec{\sigma} \models \gamma_i$ for $K_{\vec{\sigma}} \models \gamma_i$.

2.6 SRML Semantics for Strategies

Now that we have formally defined strategy profiles and their outcome, we show how any strategy σ_i for a player *i* can be specified by an SRML module $m(\sigma_i)$, and hence how we can specify any strategy profile $\vec{\sigma}$ by an SRML arena $A_{\vec{\sigma}}$.

Lemma 17 (From [10]) *For every strategy σ_i for a player i there is an* SRML *module $m(\sigma_i)$, of size linear in $|\sigma_i|$ such that the behaviour of $m(\sigma_i)$ is exactly as the behaviour of σ_i.*

Proof: Let $\sigma_i = (Q_i, q_i^0, \delta_i, \tau_i)$ be the strategy for a module $m_i = (\Phi_i, I_i, U_i)$. We construct $m(\sigma_i) = (\Phi_i', I_i', U'i)$, the module specifying σ_i. For each state $q \in Q_i$ we introduce a fresh Boolean variable that we will also denote by q, hence the module $m(\sigma_i)$ will control variables in $\Phi_i' = \Phi_i \cup Q_i$.

Our module has a single **init** command, where $\tau_i(q_i^0) = \{x_1, ..., x_k\}$:

$$I_i' = \{\top \rightsquigarrow q_i^0 := \top, x_1 := \top, ..., x_n := \top\}$$

We then use the **update** commands to encode both the output function τ_i and the transition relation δ_i. It contains, for all valuations $v_i \in V_i$ and $v_{-i} \in V_{-i}$, and all states $q, q' \in Q_i$ such that $\tau_i(q) = v_i$ and $q' \in \delta_i(q, v_{-i})$, a guarded command given by:

$$\mathcal{X}_v \wedge q \rightsquigarrow q := \bot, q' := \top, x_1 := \top, ..., x_n := \top, y_1 := \bot, ..., y_n := \bot$$

if $q \neq q'$, and otherwise:

$$\mathcal{X}_v \wedge q \rightsquigarrow q := \top, x_1 := \top, ..., x_n := \top, y_1 := \bot, ..., y_n := \bot$$

where $\{x_1, ..., x_n\} = \tau_i(q')$ and $\{y_1, ..., y_n\} = \Phi_i \backslash \tau_i(q')$, and $\mathcal{X}_v = \bigwedge_{x \in v} x \wedge \bigwedge_{x \notin v} \neg x$ is the characteristic formula for valuation $v = v_i \cup v_{-i}$.

We observe that $m(\sigma_i)$ has a single **init** command, and one **update** command for each transition in δ_i. Hence, $|m(\sigma_i)|$ is linear in $|\sigma_i|$. By construction, we immediately get that the behaviour of $m(\sigma_i)$ is exactly the same as that of σ_i. ∎

We can now extend this to the specification of a profile of strategies $\vec{\sigma}$ for an arena $A = (N, \Phi, m_1, ..., m_n)$. Once every σ_i is specified by an SRML module $m(\sigma_i)$, the outcome of $\vec{\sigma}$ is given by the Kripke structure induced by the SRML arena:

$$A_{\vec{\sigma}} = (N, \Phi \cup \bigcup_{i \in N} Q_i, m(\sigma_1), ..., m(\sigma_n)$$

Hence, the following corollary immediately follows from Lemma 11.

Corollary 18 *Let $A = (N, \Phi, m_1, ..., m_n)$ be an* SRML *arena. Then, for every strategy profile $\vec{\sigma}$ for A with outcome $K_{\vec{\sigma}}$, there is an arena $A_{\vec{\sigma}}$, of size linear in $|\vec{\sigma}|$, such that:*

$$\mathbf{trees}(K_{\vec{\sigma}}) = \mathbf{trees}(A_{\vec{\sigma}})$$

Therefore, a strategy profile $\vec{\sigma}$ can equally be specified as an SRML arena $A_{\vec{\sigma}}$. Note, however, that the outcome of this new arena will be defined with respect to valuations in $\Phi \cup \bigcup_{i \in N} Q_i$ rather than to simply $V(\Phi)$. Nevertheless, we can define *the restriction of $K_{\vec{\sigma}}$ to* Φ to be the Kripke structure $K_{\vec{\sigma}}|\Phi = (S, S^0, \delta, \pi|\Phi)$ where the valuation function is redefined as $\pi|\Phi(s) = \pi(s) \cap \Phi$.

We can now apply the logic-based semantics for SRML arenas to the semantics of strategy profiles. The following corollary directly follows from Lemma 12:

Corollary 19 *Given a strategy profile $\vec{\sigma}$ of size $|\vec{\sigma}|$ and with outcome $K_{\vec{\sigma}}$, there is a CTL formula* $\textsc{Th}(A_{\vec{\sigma}})$*, of size polynomial in $|\vec{\sigma}|$, such that*

$$\boldsymbol{\kappa} \in \mathbf{trees}(K_{\vec{\sigma}}) \iff K \models \textsc{Th}(A_{\vec{\sigma}})$$

for every Kripke structure K and computation tree $\boldsymbol{\kappa} \in \mathbf{unfold}(K)$.

3 Equilibrium Checking

In game theory, a *solution concept* is a formal rule for predicting what strategies players will adopt, and hence predict the outcome of a game. Such a prediction is called a *solution* for the game. Intuitively, a solution concept gives a formal definition of what we mean when we say that players "act rationally".

In the case of two-player *zero-sum* games (in which the gain of one player is exactly the loss of the other, and hence both sum to zero), the pioneering work of Morgenstern and von Neumann [14] gives a definite answer: there is a unique solution that is rational for both players (assuming that players have perfect information of the game), and it can be computed using the famous Minimax theorem.

The main insight in Nash's Ph.D. thesis [15] was that when we consider non-zero-sum games, we cannot predict the choices of the players if we consider them in isolation. Instead, we need to consider what strategy a player should adopt, *taking into account the strategy of the other players*. Thus, he introduced the idea of an *equilibrium* point for a game: a profile of strategy such that, given the strategy of the other players, each player has no incentive to deviate from its own.

Various equilibrium concepts will yield various definitions of what is meant by "no incentive to deviate", based on different assumptions about the players' rationality. In this project, we will focus on the most commonly used solution concept — namely *Nash equilibrium*.

3.1 Nash Equilibria in RMG

We are now in a position to define what it means for a strategy profile $\vec{\sigma}$ to be a *Nash Equilibrium for a game* $G = (A, \vec{\gamma})$. We first define, for each player i with goal γ_i, a *preference relation* $\succcurlyeq_i$ over strategy profiles:

$$\vec{\sigma} \succcurlyeq_i \vec{\sigma}' \quad \text{if and only if} \quad (\vec{\sigma}' \models \gamma_i \text{ implies } \vec{\sigma} \models \gamma_i)$$

Intuitively, a strategy profile is a Nash Equilibrium if no player can do better by changing its strategy unilaterally, i.e. given the strategies of the other players, each player is playing an optimal strategy. Formally, we have:

Definition 20 *Given an RMG* $G = (A, \vec{\gamma})$*, a strategy profile* $\vec{\sigma} = (\sigma_1, ..., \sigma_n)$ *is a Nash Equilibrium for G if, for all players i and all strategies* σ_i' *consistent with* m_i*, we have*

$$\vec{\sigma} \succcurlyeq_i (\sigma_1, ..., \sigma_{i-1}, \sigma_i', \sigma_{i+1}, ..., \sigma_n)$$

We denote by $NE(G)$ *the set of strategy profiles that are Nash Equilibria for G.*

This definition naturally leads to several game-theoretic decision questions for RMG: does a given game have at least one Nash equilibrium (NON-EMPTINESS)? For a given game, is a given CTL formula satisfied on some Nash equilibrium (E-NASH)? On all Nash equilibria (A-NASH)?

For a comprehensive survey of RMG decision problems and their associated complexity results, see GUTIERREZ [10].

3.2 The Equilibrium Checking Algorithm

We can now introduce the decision problem that is the main topic of this project:

NE-MEMBERSHIP
Given: RMG G and strategy profile $\vec{\sigma}$.
Question: Is it the case that $\vec{\sigma} \in NE(G)$?

Theorem 21 (From [10]) NE-MEMBERSHIP $\in$ EXPTIME.

Proof: We present an algorithm that decides NE-MEMBERSHIP in time at most exponential in the size of the game G and of the strategy profile $\vec{\sigma}$:

Membership$(G, \vec{\sigma})$ **where** $G = (A, \vec{\gamma}),\ A = (N, \Phi, m_1, \ldots, m_n)$
1. **for** every player $i \in N$:
2. **if** $\vec{\sigma} \not\models \gamma_i$:
3. **if** $(\gamma_i \wedge \text{TH}(A_i)) \in$ CTL SAT:
where $A_i = (N, \Phi \cup \bigcup_{j \in N \setminus \{i\}} Q_j, m(\sigma_1), ..., m_i, ..., m(\sigma_n))$
4. **reject**
5. **accept**

3.3 Correctness

The algorithm works by first checking for every player i whether its goal is satisfied: this amounts to checking whether the Kripke structure $K_{\vec{\sigma}}$ induced by $\vec{\sigma}$ satisfies the formula γ_i (line 2). Then for every player that did not get its goal achieved, it checks whether the player could have changed its strategy to satisfy its goal (line 3).

In order to check this, the algorithm first builds the SRML arena A_i, where every module m_j with $j \in N \setminus \{i\}$ is replaced by $m(\sigma_j)$, following the construction in Lemma 17. A computation tree of A_i therefore corresponds to player i not selecting any particular strategy, while every other player $j \in N \setminus \{i\}$ plays according to its strategy σ_j.

Then, using Lemma 12, the algorithm creates the CTL theory $\text{TH}(A_i)$ for this new arena, such that the structures induced by A_i are exactly those satisfying $\text{TH}(A_i)$. Finally, it checks whether the conjunction of this theory and of the goal of player i is a satisfiable CTL formula, and if so, it rejects.

Hence, the algorithm *rejects* if and only if there is some player that did not get its goal achieved and that could change its strategy so it does, i.e. if and only if $\vec{\sigma}$ is *not* a Nash equilibrium for the RMG G.

3.4 Complexity

Firstly, in order to check $\vec{\sigma} \not\models \gamma_i$, we need to construct the Kripke structure $K_{\vec{\sigma}}$ induced by $A_{\vec{\sigma}}$. From Lemma 17, the arena $A_{\vec{\sigma}}$ is of size linear in $\vec{\sigma}$, and from Lemma 11 $K_{\vec{\sigma}}$ may be at most exponential in the size of $A_{\vec{\sigma}}$. Therefore the size of the Kripke structure $K_{\vec{\sigma}}$ is at most exponential in the size of the strategy profile $\vec{\sigma}$. From [8], we get that CTL model checking over Kripke structures can be done in polynomial time. Hence, checking whether a player i gets its goal γ_i achieved can be done in EXPTIME.

Then, checking whether a player can switch to an alternative strategy and get its goal achieved amounts to deciding the satisfiability of $\gamma_i \wedge \textsc{Th}(A_i)$. From Lemma 12 and Lemma 17, we get that this formula is at most polynomial in the size of the module m_i and the strategy profile $\vec{\sigma}$. From [8], we get that checking satisfiability of CTL formulae can be done in exponential time. Hence, this step can also be done in EXPTIME.

We go through the **for** loop at most N times, hence NE-MEMBERSHIP $\in$ EXPTIME. ∎

It is possible to prove that this bound is tight. Indeed, for a proof that NE-MEMBERSHIP is EXPTIME-hard see [10].

4 Case Study: A Peer-to-Peer Example

We now present a case study based on the example presented in the introduction of [9]. Consider a peer-to-peer network with only two agents. At each time step, each agent either tries to download or to upload. In order for one agent to download successfully, the other must be uploading at the same time, and both are interested in downloading infinitely often.

While [9] considers iterated Boolean Games (iBG), where there are no constraints on the values players choose for the variables under their control, we will consider a modified version of the game: using guarded commands, we require that a player cannot both download and upload at the same time. This is a simple example of a game which cannot be specified as an iBG, but which has an SRML representation.

We can specify this game as an RMG with two players, 0 and 1, where each player $i \in \{0, 1\}$ controls two variables u_i ("Player i tries to upload") and d_i ("Player i tries to download"); player i downloads successfully if $(d_i \wedge u_{i-1})$.

Formally, we define $G = (A, \vec{\gamma})$ where $A = (2, \Phi, m_0, m_1)$, $\Phi = \{u_0, u_1, d_0, d_1\}$ and the modules m_0, m_1 are defined as follows:

$$\begin{array}{ll}
\textbf{module } m_0 \textbf{ controls } u_0, d_0 & \textbf{module } m_1 \textbf{ controls } u_1, d_1 \\
\quad \textbf{init} & \quad \textbf{init} \\
\quad :: \top \rightsquigarrow u_0' := \top,\ d_0' := \bot & \quad :: \top \rightsquigarrow u_1' := \top,\ d_1' := \bot \\
\quad :: \top \rightsquigarrow u_0' := \bot,\ d_0' := \top & \quad :: \top \rightsquigarrow u_1' := \bot,\ d_1' := \top \\
\quad \textbf{update} & \quad \textbf{update} \\
\quad :: \top \rightsquigarrow u_0' := \top,\ d_0' := \bot & \quad :: \top \rightsquigarrow u_1' := \top,\ d_1' := \bot \\
\quad :: \top \rightsquigarrow u_0' := \bot,\ d_0' := \top & \quad :: \top \rightsquigarrow u_1' := \bot,\ d_1' := \top
\end{array}$$

In the context of CTL RMGs, *"infinitely often"* will be expressed as *"on all paths, always, on all paths, eventually"*. Hence, for $i \in \{0, 1\}$ we define $\gamma_i = \mathbf{AGAF}(d_i \wedge u_{1-i})$.

4.1 A simple Nash equilibrium

A solution to this game would be for the players to alternate: on one round player i downloads and player $(1 - i)$ uploads, then on the next round player $(1 - i)$ downloads and player i uploads. This would ensure that both players download infinitely often and hence that both get their goal achieved.

For instance, consider the following strategy profile $\vec{\sigma}$, modelled by the SRML specification $m(\vec{\sigma}) = (2, \Phi, \mathit{first}, \mathit{second})$, where modules *first* and *second* are defined as follows:

$$\begin{array}{ll}
\textbf{module } \mathit{first} \textbf{ controls } u_0, d_0 & \textbf{module } \mathit{second} \textbf{ controls } u_1, d_1 \\
\quad \textbf{init} & \quad \textbf{init} \\
\quad :: \top \rightsquigarrow u_0' := \top,\ d_0' := \bot & \quad :: \top \rightsquigarrow u_1' := \bot,\ d_1' := \top \\
\quad \textbf{update} & \quad \textbf{update} \\
\quad :: \top \rightsquigarrow u_0' := d_0,\ d_0' := u_0 & \quad :: \top \rightsquigarrow u_1' := d_1,\ d_1' := u_1
\end{array}$$

Player 0 will be the first to upload, and player 1 the first to successfully download, then they alternate at every round. Hence, they will both get their goal achieved and therefore have no incentive to change their strategy: this is a Nash equilibrium.

However, this strategy profile requires the players to communicate before the game and agree on which player adopts the strategy *first*, and which player adopts *second*. In the following examples, we discuss *symmetrical* strategy profiles, in which this kind of prior communication is not possible.

4.2 TITFORTAT

Suppose that the players have no means of communication prior to the game. We assume that both players are equally rational, and hence that they will reason the same way and adopt symmetrical strategies.

Suppose further that both players are willing to cooperate, but only if the other player does as well. They could adopt the following TITFORTAT strategy (adapted from [9]): if I successfully downloaded at the previous round, I will upload at the next; and if the other player successfully downloaded at the previous round, I will download at the next.

The crucial difference between the game discussed in [9] and our modified version is that here, if both players decide to upload first then neither of them will succeed in downloading. So suppose that they both start by non-deterministically choosing whether to upload or download, and that if they happen to have made the same choice (i.e. $(u_i \wedge u_{i-1})$ or $(d_i \wedge d_{i-1})$), then they will keep on making a non-deterministic choice.

This strategy can be specified with the following SRML module:

$$
\begin{array}{l}
\textbf{module } \textsc{TitForTat}(i) \textbf{ controls } u_i, d_i \\
\quad \textbf{init} \\
\quad :: \top \rightsquigarrow u_i' := \top,\ d_i' := \bot \\
\quad :: \top \rightsquigarrow u_i' := \bot,\ d_i' := \top \\
\quad \textbf{update} \\
\quad :: (u_{i-1} \wedge d_i) \rightsquigarrow u_i' := \top,\ d_i' := \bot \\
\quad :: (d_{i-1} \wedge u_i) \rightsquigarrow u_i' := \bot,\ d_i' := \top \\
\quad :: (u_i \wedge u_{i-1}) \vee (d_i \wedge d_{i-1}) \rightsquigarrow u_i' := \bot,\ d_i' := \top \\
\quad :: (u_i \wedge u_{i-1}) \vee (d_i \wedge d_{i-1}) \rightsquigarrow u_i' := \top,\ d_i' := \bot
\end{array}
$$

However, both players following this same strategy will result in an outcome that satisfies none of their goal. Indeed, there will be some paths where the players never reach the desired alternation: if both players keep on making the same non-deterministic choices, then none of them can successfully download.

It is easy to see that any of the players could change its strategy to get its goal achieved, for example by following the strategy specified by the *toggle* module in Section 2.1, with x for u_i and y for d_i. Hence, this cannot be a Nash equilibrium.

4.3 BLOCK

Interestingly, it is possible to prove that, if we assume that the players follow symmetrical strategies (as is the case for TITFORTAT), then such a strategy profile is a Nash equilibrium if and only if it prevents both players to satisfy their goal, as is discussed in the following example.

Suppose that both players decide that they do not want to cooperate at all, as specified in the following SRML module:

$$
\begin{array}{l}
\textbf{module } \textsc{Block}(i) \textbf{ controls } u_i, d_i \\
\quad \textbf{init} \\
\quad :: \top \rightsquigarrow u_i' := \bot,\ d_i' := \top \\
\quad \textbf{update} \\
\quad :: \top \rightsquigarrow u_i' := \bot,\ d_i' := \top
\end{array}
$$

Obviously in this case, neither player's goal will be achieved. However, neither player can benefit from changing their strategy unilaterally, because whatever they do, the other player prevents them from achieving their goal. Hence, this is a Nash equilibrium.

It can seem counter-intuitive that this should be considered as a solution for the game. Indeed, from a third-person perspective, both players would benefit from changing their strategy: this strategy profile is not *Pareto optimal* (i.e. it is possible to change the outcome so that at least one player is better off while no other player is worse off).

Some solution concepts refine the concept of a Nash equilibrium to eliminate those kind of undesirable behaviours. For example, in a *strong Nash equilibrium* no coalition of players can cooperate and, given the strategies of the other players, change their strategy in a way that benefits all the members of the coalition. It is possible to prove that a strong Nash equilibrium is necessarily Pareto optimal.

4.4 Conclusions — Symmetry in Coordination Games

We have presented a simple example of a reactive module game, and three examples of strategy profiles which show the three possible cases in the equilibrium analysis. First, an example of a Nash equilibrium where both players' goals are achieved. Then, an example where neither player gets its goal achieved, and they could change their strategy so they do: this is not a Nash equilibrium. Finally, an example where neither player's goal is achieved, but both players prevent the other from the possibility of a beneficial change of strategy, hence this is a Nash equilibrium.

This simple example is an instance of a class of theoretical games called *coordination games*, in which players can realize mutual gains, but only by adopting mutually consistent strategies. The applications of coordination games are widespread in social sciences, ranging from the problem of choosing which side of the road upon which to drive to the choice of technological standards.

We have seen the crucial role of symmetry in game theory: in our example, assuming that the players adopt symmetrical strategies leads to the counter-intuitive conclusion that they should not cooperate. Exploiting symmetry in games allows for more compact representation and it often provides algorithmic short-cuts leading to more efficient procedures for computing equilibria. The study of symmetric games was first considered in Nash's original paper [15], but it is an area of ongoing research. For an overview of equilibria in symmetric games, see [4].

Conclusions and Future Work

In this project, we have investigated the properties of *computational economies*: multi-agent systems in which each agent has its own goals/preferences about the overall behaviour of the system, and where these agents are assumed to adopt selfish strategies in order to achieve their goal. This led us to explore concepts at the intersection of distributed artificial intelligence, game theory, computational complexity, formal verification and logic.

We have made a thorough presentation of the formalism introduced in [10], namely *Reactive Modules Games*, which model the behaviour of rational agents in concurrent multi-agent systems. In particular, we focused on the problem of *equilibrium checking* — checking whether a set of strategies is a *Nash equilbrium* for a given game; and we gave an improved presentation of the proof that it belongs to the EXPTIME complexity class.

We have successfully implemented a tool for solving this problem, which will be a preliminary proof-of-concept for the RACE system (*Reasoning About Computational Economies*): a robust model checking tool for the formal specification, verification and analysis of these systems. Our results are presented in Appendices B and C.

In the course of this implementation, we revised the proof of the lemma which gives a CTL-based semantics for RMG arenas (Lemma 12 of this report), providing a new construction which does not make use of the CTL existential path quantifier, and hence belongs to the common fragment of CTL and LTL. In Appendix A we present the reasoning that led to this new proof, as well as the consequences it has on our implementation.

Future Work

Even though our first results are promising, the system that we implemented was meant to be a proof of concept more than a practical tool. Indeed, there is much work to be done before it can be scaled to more realistic scenarios than the simple examples we considered in this project.

As discussed in Appendix B, the major limiting step in the algorithm was to check the satisfiability of CTL formulae, a problem for which the worst-case is exponential in the size of the formula. This became intractable for more than two-player games, at least a standard laptop. From a complexity standpoint, our problem is EXPTIME-*hard* [10]. Hence, from the time hierarchy theorem we know that there is no polynomial-time solution for the general case.

However, we believe that in practice this is the step where efficiency could be improved most. The SAT solving tool we had at our disposal was highly inefficient, and we think that our system would gain a lot from a dedicated solver. Indeed, it would be possible to implement heuristics that are specific to the CTL construction presented in Lemma 12, and thus improve the average-case running time. In Appendix A, we discuss one of the possible improvements along this line.

On the more theoretical side, the theory of Reactive Modules Games could be extended in several directions. In the case study we presented in Section 4, we showed how Nash equilibria can yield counter-intuitive solutions. It would be very interesting to investigate other solution concepts, and how they affect the complexity of equilibrium check-

ing. Refinements to the concept of Nash equilibrium are abundant in the game theory literature (see, e.g. strong Nash equilibrium, sub-game perfect and secure equilibrium).

A second line of research would be to explore the use of other temporal logic languages for expressing player's goals, and to investigate their associated complexity results. We could consider fragments of CTL which would yield more tractable complexity results, (for instance, the common fragment of CTL and LTL which we discuss in Appendix A). On the other hand we could consider more expressive logics — e.g. CTL* and the μ-calculus, which contain both CTL and LTL. Yet another line would be to consider altogether different logics — see, e.g. Moszkowski's *Interval Temporal Logic*, which deals with finite time intervals instead of infinite sequences of state.

Finally, Reactive Modules Games could be extended to deal with different classes of theoretical games. In the current formalism we assume that players have access to all information relevant to making their decisions; removing this assumption would yield games with *imperfect information*, of which Texas hold'em is the most famous example. The study of temporal logics to reason about games with imperfect information is an area of ongoing research — see, e.g. Hintikka and Sandu's *Independence-Friendly* (IF) logic and the variant developped in [5].

Another assumption is that the outcome of a game is binary: either a player achieves its goal or it does not. Removing this assumption to study more general payoff sets would allow to study a more general class of games — for instance the ubiquitous *Prisoner's Dilemma*, for which we need to be able to distinguish between a 1-year and a 10-year prison sentence. One way to achieve this would be to associate each player with more than one goal, and to associate different goals with different rewards.

Personal Conclusions

Overall, I am very pleased with the outcomes of this project.

It allowed me to study theoretical concepts and practical techniques that are outside the taught part of my curriculum: the analysis of multi-agent systems, formal verification and model checking, computation tree logic and its associated decision questions, and the methods required for proving complexity results about these problems.

Moreover, I had the opportunity to familiarise myself with the main concepts and insights of game theory — which has become a major source of personal interest — and to explore the interface it shares with theoretical computer science and artificial intelligence.

Finally, and perhaps most importantly, this project has given me a real taste of what academic research is like. It has taught me some of the patience and the motivation required to complete this first long-term project, but also to plan and to write this report.

Acknowledgements

I would like to use this last sentence to thank my supervisor Julian Gutierrez for the kind help he has provided me throughout this project, as well as in the writing of this report.

Appendix

A CTL Semantics for SRML

Our proof of Lemma 12 is the main contribution of this project to the theory of Reactive Modules Games. Indeed, our construction of a CTL theory for SRML arenas corrects the issues we found in the original construction presented in [10].

Our first implementation of the membership algorithm presented in Section 3.2 relied on this original construction. When testing our system, we arrived at counter-intuitive results. This led us to revise the construction and to arrive at the proof we present in Section 2.4.

In this appendix, we will describe one example of an input for which the algorithm gave an incorrect answer, and then explain what went wrong in the construction of the CTL theory for this input. First, we include a copy of the original construction:

> Let $A = (N, \Phi, m_1, ..., m_n)$ be the given arena. We will define $\text{TH}(A)$ as the conjunction of three predicates:
>
> $$\text{TH}(A) = \text{POL} \wedge \text{INIT}(A) \wedge \text{UPDATE}(A)$$
>
> First, we introduce two sets of Boolean variables Φ^+ and Φ^- such that
>
> $$x^+ \in \Phi^+ \iff x \in \Phi \iff x^- \in \Phi^-$$
>
> We use the variables x^+ and x^- to hold, alternately, the current and the next value of variable x. To control this alternation, we introduce a fresh Boolean variable p called the *polarity* variable: whenever p is true, x^- will hold the current value of x and x^+ holds the next value of x, and it will be the opposite whenever p is false. The predicate POL asserts that the value of p alternates between true and false at every step, starting with true:
>
> $$\text{POL} = p \wedge \mathbf{AG}(p \leftrightarrow \mathbf{AX}\neg p)$$
>
> We now define the semantics for the initialisation of the modules, given by $\text{INIT}(A)$. We first define the effect of a single **init** command $g = \top \rightsquigarrow x_1' := b_1, ..., x_k' := b_k$ for module m_i:
>
> $$\text{INIT}_i(g) = \bigwedge_{x \in \Phi_i \setminus ctr(g)} (x^+ \leftrightarrow \bot \leftrightarrow x^-) \wedge \bigwedge_{1 \le l \le k} (x_l^+ \leftrightarrow b_l \leftrightarrow x_l^-)$$
>
> Then the predicate INIT_i asserts that exactly one **init** command is executed for a module $m_i = (\Phi_i, I_i, U_i)$:
>
> $$\text{INIT}_i = \bigoplus_{g \in I_i} \text{INIT}_i(g)$$

Finally, we can define the initialisation predicate for the arena A:

$$\textsc{Init}(A) = \bigwedge_{i \in N} \textsc{Init}_i$$

We then define the semantics for the **update** commands, given by $\textsc{Update}(A)$. We begin by defining the predicate $\textsc{Unch}(\Psi)$, which asserts that the values of the variables in $\Psi \subseteq \Phi$ will remain unchanged in the next state:

$$\textsc{Unch}(\Psi) = \bigwedge_{x \in \Psi} x^+ \leftrightarrow x^-$$

For each **update** command $g = \varphi \rightsquigarrow x'_1 := \psi_1, ..., x'_k := \psi_k$ for module m_i, we define:

$$\textsc{Update}_i^+(g) = \overbrace{p \wedge \varphi^-}^{(i)} \wedge \overbrace{\textsc{Unch}(\Phi_i \backslash ctr(g))}^{(ii)} \wedge \overbrace{\bigwedge_{1 \leq l \leq k} x_l^+ \leftrightarrow \psi_l^-}^{(iii)} \qquad \text{and,}$$

$$\textsc{Update}_i^-(g) = \overbrace{\neg p \wedge \varphi^+}^{(i)} \wedge \overbrace{\textsc{Unch}(\Phi_i \backslash ctr(g))}^{(ii)} \wedge \overbrace{\bigwedge_{1 \leq l \leq k} x_l^- \leftrightarrow \psi_l^+}^{(iii)}$$

where φ^+ is defined to be the formula φ, except that every occurrence of every variable $x \in \Phi$ has been replaced by x^+, and similarly for φ^-, ψ^+ and ψ^-.

Hence, for a command $g \in U_i$, the formula $(\textsc{Update}_i^+(g) \oplus \textsc{Update}_i^-(g))$ asserts that: *(i)* the current valuation satisfies $guard(g)$, *(ii)* all the variables not affected by the command remain unchanged, and *(iii)* the command is executed to obtain the values of the variables in $ctr(g)$ for the next state of the system.

Then, $\textsc{Update}_i$ asserts that *(i)* if no **update** commands are enabled, then module m_i executes the null-guarded command, and *(ii)* otherwise it executes exactly one enabled **update** command:

$$\textsc{Update}_i = \overbrace{(\textsc{Unch}(\Phi_i) \wedge \bigwedge_{g \in U_i} \neg guard'(g))}^{(i)} \vee \overbrace{\bigoplus_{g \in U_i} (\textsc{Update}_i^+(g) \oplus \textsc{Update}_i^-(g))}^{(i)}$$

where $guard'(g)$ is obtained from $guard(g)$ by replacing every occurence of $x \in \Phi$ by $(p \rightarrow x^+) \wedge (\neg p \rightarrow x^-)$.

Finally, we can define the update predicate for the arena A:

$$\textsc{Update}(A) = \mathbf{AGEX} \bigwedge_{i \in N} \textsc{Update}_i$$

which asserts that at any point in time (**AG**: "on all paths, always") there is at least one joint guarded command to get to some next state, i.e. there is some path where at the next step (**EX**) the conjunction of the $\textsc{Update}_i$ predicates is satisfied.

We now present the simplest example for which this construction turned out to be incorrect. The arena $A = (1, \{x\}, m)$ consists of a single module which controls a single variable, the module m has a single **init** command and a single **update** command.

$$\begin{array}{l}\textbf{module } m \textbf{ controls } x \\ \quad \textbf{init} \\ \quad :: \top \rightsquigarrow x' := \top \\ \quad \textbf{update} \\ \quad :: \top \rightsquigarrow x' := \neg x\end{array}$$

We ran our implementation on this example with the goal $\mathbf{AG}x$: "the value of x is always $\top$". It answered that the player could change its strategy so that this goal is achieved. Obviously this answer is incorrect: indeed the module specifies that the value of x must alternate between $\top$ and $\bot$ at every round, hence it should not be possible for the value of x to be always $\top$.

Let us now explain the details of the incorrect CTL theory for this example. First, $\textsc{Init}(A)$ asserts that the initial value of x is $\top$:

$$\textsc{Init}(A) = x^+ \leftrightarrow \top \leftrightarrow x^-$$

Then, we construct the predicates for the single **update** command:

$$\textsc{Update}_0^+ = p \wedge \top \wedge (x^+ \leftrightarrow \neg x^-) \qquad \text{and,}$$

$$\textsc{Update}_0^- = \neg p \wedge \top \wedge (x^- \leftrightarrow \neg x^+)$$

This **update** command is always enabled, hence there is no need for the null-guarded command and we have:

$$\textsc{Update}_0 = (p \wedge \top \wedge (x^+ \leftrightarrow \neg x^-)) \oplus (\neg p \wedge \top \wedge (x^- \leftrightarrow \neg x^+))$$

Which is equivalent to simply $(x^+ \leftrightarrow \neg x^-)$. Hence, $\textsc{Update}(A)$ becomes

$$\mathbf{AGEX}(x^+ \leftrightarrow \neg x^-)$$

Together with $\textsc{Init}(A)$ and $\textsc{Pol}$, this asserts that x^+ and x^- should have opposite values, but it cannot distinguish which one should be the current value and which one should be the previous value. Hence, this explains why, when put in conjunction with the goal $\mathbf{AG}((p \rightarrow x^+) \wedge (\neg p \rightarrow x^-))$, the formula thus obtained is satisfiable. Indeed, the Kripke structure $K = (S, S^0, R, \pi)$ where

$$\begin{array}{l} S = \{0, 1\} \\ S^0 = \{0\} \\ R = \{(0,1), (1,0)\} \\ \pi(0) = \{p, x^+\} \\ \pi(1) = \{\neg p, x^-\} \end{array}$$

is a model of this formula, while the lemma should imply that it is not satisfiable. Hence, this is an example for which the construction is incorrect.

In the light of this example, we arrived at the conclusion that the construction could not be correct. Here, we will explain the reasoning that led us to revise the proof of Lemma 12 and arrive at a correct construction.

We observed that the use of two separate sets of variables and of a POL predicate was motivated by the fact that the construction in [10] for the LTL case implemented the assignment statements $x := \psi$ in **update** commands with the equivalence predicate $\psi \leftrightarrow \mathbf{X}x$.

This is not possible in CTL because tense operators must be preceded by path quantifiers: $\psi \leftrightarrow \mathbf{AX}x$ would leave open the possibility that ψ is false but that there is some next state where x is true, $\psi \leftrightarrow \mathbf{EX}x$ would leave open the possibility that ψ is true but that there is some next state where x is false.

The main insight was to break down this equivalence into two implications and treat them separately, which led us to define the ASSIGN predicate:

$$\textsc{Assign}(x := \psi) = (\psi \rightarrow \mathbf{AX}x) \wedge (\neg\psi \rightarrow \mathbf{AX}\neg x)$$

Then, we were in a position to apply the same construction as the LTL case in a straightforward way, without the need of two separate sets of variables and of the POL predicate. That this construction was correct followed from the same argument as for the LTL case.

A consequence of this result is that the CTL theory for an SRML arena belongs to the common fragment of LTL and CTL. Indeed, our construction only uses universal path quantifiers. The semantics of LTL formulae imply that they are all universally quantified in an implicit way, hence removing all universal quantifiers from our construction yields an equisatisfiable LTL formula. [6]

This property has interesting consequences for the complexity of the equilibrium checking algorithm presented in Section 3.2. Indeed, whereas CTL SAT is EXPTIME-complete, LTL SAT is in PSPACE. Therefore, in the special case of goals expressed without existential quantifiers, the step of checking if $\gamma_i \wedge \textsc{Th}(A_i)$ is satisfiable (line 3) can be done in polynomial space instead of exponential time.

B Implementation Notes

One of the aims of this project was to build a proof-of-concept implementation of the equilibrium checking algorithm described in Section 3.2. In this appendix, we present the system we developed for checking, given a game G specified in the SRML language and a profile of strategies $\vec{\sigma}$ for this game, whether the strategy profile $\vec{\sigma}$ is a Nash equilibrium for G.

We first present the requirement specification for the system, then we describe the design decisions that were taken to meet these requirements, and we present the architecture of the system.

Specification

Our system was required to input an RMG $G = (A = (N, \Phi, m_0, ..., m_n), \vec{\gamma})$ and a strategy profile $\vec{\sigma}$ such that:

1. The reactive modules m_i should be input in a human-readable format that is close to the concrete syntax presented in Section 2.1.

2. The CTL formulae $\vec{\gamma}$ expressing the goals of the players should simply be strings.

3. Following Lemma 17, the strategy profile $\vec{\sigma}$ should be input as a list of reactive modules, hence using the same format as specified in 1.

A command-line interface which can input text files was considered sufficient. Moreover, the system should successfully implement the algorithm described in Section 3.2, that is:

4. On input $(G, \vec{\sigma})$, it should output **True** if and only if $\vec{\sigma} \in NE(G)$.

5. It should halt on all correct inputs (see assumptions), and run in time at most exponential in the size of its input, as argued in Section 3.4.

We also required that the command-line interface implements a "verbose" mode, in which it gives a detailed account of the running of the algorithm: whether players get their goal achieved, and in the case they do not, whether they could benefit from changing their strategy. It should also give measurements of the running time of the algorithm, which will be used in evaluating the performances of our implementation.

We made the following assumptions, which define what a correct input is. We required

1. That the input modules, both in the arena and in the strategy profile, respect the specification of SRML . In particular, we required:

   (a) That no variable is assigned twice in the same guarded command.
   (b) That the guards to **init** commands are "$\top$".

(c) That in the assignment statements $x := \psi$ in **init** commands, ψ is a Boolean constant, $\top$ or $\bot$.
(d) That for every input module $m_i = (\Phi_i, I_i, U_i)$, I_i and U_i are sets instead of bags, i.e. that they contain only pairwise distinct elements.
(e) That for every input module $m_i = (\Phi_i, I_i, U_i)$ and for every command $g \in I_i \cup U_i$ we have that $ctr(g) \subseteq \Phi_i$.

2. That the input strings for goals are syntactically correct CTL formulae, in particular that they respect the alternation between path quantifiers and tense operators.

3. That the input strategy profile is consistent with the input arena, as discussed in Remark 15.

Design Decisions

Choice of programming language and tools. The first step in the implementation of the algorithm was to choose an appropriate programming language. We first considered using Haskell: pattern matching over algebraic data types seemed to be the most intuitive way to work with the syntax trees of CTL formulae. Moreover, the purely functional paradigm of Haskell allowed for a straightforward translation of the formal definition of reactive modules.

However, we needed to find external tools for the CTL SAT and CTL MC black boxes (presented in Section 1.4 and 1.5), and this turned out to make Haskell impractical. Indeed, we decided to use the Python CTL model checker MR.WAFFLES [18], and hence we chose Python as the main programming language of our system. Finding an implementation of the algorithm for CTL SAT turned out to be even more difficult. Indeed, CTLSAT [17] was the only such tool available freely on the Internet.

MR.WAFFLES implements Kripke structures with a class `PredicatedGraph` which extends the `networkx` library for graphs with a `predicate` attribute for every node: a list of the propositional variables (represented as strings) that are true at this node. It then provides a `check` method that takes a string representing a CTL formula (in prefix notation) and outputs a list of the states at which the formula is satisfied. Hence, checking whether the Kripke structure satisfies the formula amounts to checking that all the initial states are in this list.

CTLSAT is a command-line interface that inputs CTL formulae as strings (in infix notation) and then prints out some information about the working of the tableau algorithm and finally prints either `Input formula is satisfable!` or `Input formula is NOT satisfable!`. One of the problems we encountered was that its parser was seriously restricted: propositional atoms could be represented only as a single ascii characters, excluding the characters used for the syntax of formulae and the special characters reserved by the parser. Hence, it was limited to at most 70 distinct variables. Although a serious defect, this was considered sufficient for our purpose.

Concrete data structures. We decided to represent propositional variables as ints, and propositional valuations as lists of ints. We implemented a Python class for propositional logic, which we used to store the guards and the Boolean values of guarded commands.

There is one subclass for each case in the grammar and two special instances, `T` and `F`, to represent $\top$ and $\bot$. Every instance of the class has a `type` attribute to indicate which case the instance belongs to, and an `eval` function which inputs a valuation and outputs **True** if and only if the valuation satisfies the formula.

We implemented assignment statements as Python named tuples `(var,b)` where `var` is an int and `b` is an instance of the propositional logic class. Guarded commands are implemented as named tuples `(guard,action)` where `guard` is an instance of the propositional logic class and `action` is a list of assignment statements. Reactive modules were also implemented as named tuples `(ctrl,init,update)` where `ctrl` is a list of ints representing the variables the module controls, `init` and `update` are lists of guarded commands.

Input format. We decided that the simplest format for the input was to use Python files, which we then parsed using the Python `eval` function. The input to the equilibrium checking algorithm is represented as a Python `dict` with three keys: *(i)* `modules` is a list of reactive modules representing the SRML arena, *(ii)* `goals` is a list of CTL formulae represented as strings in MR.WAFFLES notation, and *(iii)* `strategies` is a list of reactive modules representing the strategy profile.

To input the reactive modules and their guarded commands directly as named tuples would have made the input files unnecessarily long and unreadable. Hence, we decided to represent modules as Python dictionaries with three keys: `ctrl` is the list of ints representing the controlled variables, `init` and `update` are lists of guarded commands that we decided to represent as strings, close to the concrete syntax presented in Section 2.1. The guards and the Boolean values in guarded commands are expressed using MR.WAFFLES prefix notation, and the propositional variable represented by the int `n` is simply denoted by `xn`.

Here is an example of the text for an input module, representing the *toggle* example:

```
{ # module "toggle"
'ctrl': [0, 1],
'init': [
        "T -> x0' := T, x1' := F",
        "T -> x0' := F, x1' := T"],
'update': [
        "T -> x0' := T, x1' := F",
        "T -> x0' := x1, x1' := x0"]
}
```

System architecture. Our system consists of five Python modules:

`main.py` implements the command-line interface and the main algorithm. It also implements the verbose mode and prints the running time measurements.

`pl.py` implements the propositional logic class.

`rm.py` implements the concrete data structures described in Section B, as well as the parsing of input modules and guarded commands.

`arena2kripke.py` implements the algorithm described in Figure 1. It translates an SRML arena, represented as a list of modules, to its induced Kripke structure, represented as MR.WAFFLES `PredicatedGraph` instance.

`thctl.py` implements the construction presented in Section 2.4. It translates an SRML arena, represented as a list of modules, to its associated CTL theory, represented as a string in CTLSAT notation. It is also responsible for wrapping the CTLSAT command-line interface, using Python `subprocess` module.

Testing and Performances

We tested our system on the the peer-to-peer example presented in Section 4, running it for the three strategy profiles that we considered, which were examples of the three possible cases in the analysis of Nash equilibria.

To make the analysis more tractable we used only a single variable per player: `x0` being true means that d_0 is $\top$ and u_0 is $\bot$. The SRML arena was represented as a list of two reactive modules:

```
'modules': [
        { # module m_0
        'ctrl': [0],
        'init': [
                "T -> x0' := T",
                "T -> x0' := F"],
        'update': [
                "T -> x0' := T",
                "T -> x0' := F"],
        },

        { # module m_1
        'ctrl': [1],
        'init': [
                "T -> x1' := T",
                "T -> x1' := F"],
        'update': [
                "T -> x1' := T",
                "T -> x1' := F"]
        }
]
```

and the goals were represented as strings in MR.WAFFLES infix notation:

```
'goals': ["AG AF (and x0 !x1)",  "AG AF (and !x0 x1)"]
```

A simple Nash equilibrium.

Input

```
'strategies': [
        { # first
        'ctrl': [0],
        'init': [
                "T -> x0' := F"],
        'update': [
                "T -> x0' := !x0"]
        },

        { # second
        'ctrl': [1],
        'init': [
                "T -> x1' := T"],
        'update': [
                "T -> x1' := !x1"]
        }
]
```

Output

```
Building the outcome ...
2 nodes and 2 edges. (5.0 ms)

Player 0 gets its goal achieved.
Player 1 gets its goal achieved.

        This is a Nash Equilibrium.

Model checking:          255.0 ms
CTL SAT solving:         0.0 ms
```

TITFORTAT

Input

```
'strategies': [
        { # player 0
        'ctrl': [0],
        'init': [
                "T -> x0' := T",
                "T -> x0' := F"],
        'update': [
                "and x0 !x1 -> x0' := F",
```

```
                "and !x0 x1 -> x0' := T",
                "or (and x0 x1) (and !x0 !x1) -> x0' := F",
                "or (and x0 x1) (and !x0 !x1) -> x0' := T"]
        },

        { # player 1
        'ctrl': [1],
        'init': [
                "T -> x1' := T",
                "T -> x1' := F"],
        'update': [
                "and x0 !x1 -> x1' := T",
                "and !x0 x1 -> x1' := F",
                "or (and x0 x1) (and !x0 !x1) -> x1' := F",
                "or (and x0 x1) (and !x0 !x1) -> x1' := T"]
        }
]
```

Output

```
Building the outcome ...
4 nodes and 10 edges. (12.0 ms)

Player 0 does not get its goal satisfied.

SAT solving ...
It could change its strategy to get its goal satisfied:

        Not a Nash Equilibrium.

Model checking:         128.0 ms
CTL SAT solving:        2070.0 ms
```

BLOCK

Input

```
'strategies': [
        { # player 0
        'ctrl': [0],
        'init': [
                "T -> x0' := T"],
        'update': [
                "T -> x0' := T"]
        },
```

```
        { # player 1
        'ctrl': [1],
        'init': [
                "T -> x1' := T"],
        'update': [
                "T -> x1' := T"]
        }
]
```

Output

```
Building the outcome ...
1 nodes and 1 edges. (6.0 ms)

Player 0 does not get its goal satisfied.

SAT solving ...
But it cannot change its strategy to satisfy it.

Player 1 does not get its goal satisfied.

SAT solving ...
But it cannot change its strategy to satisfy it.

        This is a Nash Equilibrium.

Model checking:          260.0 ms
CTL SAT solving:         219.0 ms
```

Performances.

Performances were difficult to evaluate for inputs in which some player does not get its goal achieved. Indeed, the fact that CTL SAT is EXPTIME-complete made any attempt at analysing non-equilibrium strategy profiles for games with more than two players intractable on a standard laptop. Nevertheless, in Appendix A we consider how, in the special case of goals without existential quantifiers, the step of checking whether a player can benefit from changing its strategy can be improved to run in PSPACE.

Nevertheless, we were able to measure the performances of our system in cases where every player gets its goal satisfied. For this, we considered the ring-based mutual exclusion algorithm presented in [10], which was simple to scale to the n-player case:

n	Parsing (ms)	Building the outcome (ms)	Model checking (ms)
2	202	15	151
4	376	35	306
8	735	87	624
16	1481	302	1252
32	3005	1683	2526
64	6063	11861	5111
128	11810	92255	10458
256	23599	825148	21465

As argued in Section 3.4, we observe that building the outcome is exponential in the number of variables, and hence in the number of players (it roughly squares when n doubles). We can observe that parsing and model checking run in time polynomial in the size of the size of the input, in this example it is even linear.

C Code

main.py

```
# This is the main module of the project.
# It defines the concrete data structures for Reactive
    Module Games, along with utility functions.
# It implements the command line interface for calling the
    algorithm, and the parsing of input files.

import time, sys, argparse, pickle

from rm import *
from arena2kripke import *
from thctl import *

# the main algorithm
def Membership(modules, goals, strategies, verbose):
    assert(len(modules)==len(goals))
    assert(len(goals)==len(strategies))

    N = len(modules)

    if verbose:
        start = time.time()*1000
        print "Building the outcome ..."

    (A, S0) = Arena2Kripke(strategies)

    if verbose:
        perfA2K = time.time()*1000 - start
        print str(A.number_of_nodes())+" nodes and "+str(A.
            number_of_edges())+" edges. ("+str(round(perfA2K))
            +" ms)"
        print

    # time counters for model checking and SAT solving
    perfMC = 0.0
    perfSAT = 0.0

    for i in range(N):
        start = time.time()*1000 # timestamp for measuring
            model checking performances

        if not entails(A, S0, goals[i]):
```

```
        perfMC += time.time()*1000 - start

        theory = Th([modules[i]]+strategies[:i]+strategies
            [i+1:])
        goal = ast2ctl(CTL.parse(goals[i]))

        print "Player "+str(i)+" does not get its goal
            satisfied."
        if verbose:
            print
            print "SAT solving ..."

        start = time.time()*1000 # timestamp for measuring
             SAT solving performances

        if SAT(goal+"^"+theory, verbose):
            perfSAT += time.time()*1000 - start

            print "It could change its strategy to get its
                goal satisfied:"
            print
            print "\tNot a Nash Equilibrium."

            if verbose:
                print
                print "Model checking: \t"+str(round(perfMC)
                    )+" ms"
                print "CTL SAT solving: \t"+str(round(
                    perfSAT))+"ms"

            return False
        else:
            perfSAT += time.time()*1000 - start
            print "But it cannot change its strategy to
                satisfy it."
    else:
        perfMC += time.time()*1000 - start
        print "Player "+str(i)+" gets its goal achieved."
print
print "\tThis is a Nash Equilibrium."

if verbose:
    print
    print "Model checking: \t"+str(round(perfMC))+" ms"
    print "CTL SAT solving: \t"+str(round(perfSAT))+" ms"
```

```
    return True

def main(argv=None): # parses the command line arguments
    and calls Membership
    argparser = argparse.ArgumentParser(description='CTL RMG
        Equilibrium Checker')
    argparser.add_argument('inputf')
    argparser.add_argument('-v', '--verbose', action="
        store_true", default=False)
    args = argparser.parse_args(sys.argv[1:])

    f = open(args.inputf)
    d = eval(f.read())

    modules = [parseRM(m) for m in d['modules']]
    strategies = [parseRM(m) for m in d['strategies']]

    Membership(modules, d['goals'], strategies, args.verbose
        )

if __name__ == '__main__':
    main()
```

rm.py

```
# This module defines the concrete data structures for
    Reactive Module Games.
# It implements utility functions, and the parsing of input
    files.

from mrwaffles.ctl import *
from pl import *

from collections import namedtuple
from itertools import product

# defines an action "var := b"
class Action(namedtuple('Action', ['var', 'b'])):
    def __repr__(self):
        return "x"+repr(self.var)+" := "+repr(self.b)

# defines a command "guard -> var1 := b1, var2 := b2, ..."
class Command(namedtuple('Command', ['guard', 'actions'])):
    def __repr__(self):
        return repr(self.guard)+" -> "+repr(self.actions)

    def __init__(self, guard, actions):
        self.ctrl = [a.var for a in self.actions]

# defines a module "m = (Phi, I, U)"
class RM(namedtuple('RM', ['ctrl', 'init', 'update'])):
    def __repr__(self):
        print "m controls "+str(self.ctrl)
        print "init"
        for g in self.init:
            print "\t"+str(g)
        print "update"
        for g in self.update:
            print "\t"+str(g)
        print enabled_init(self)

# returns the valuation obtained from the execution of
    command g on valuation v in module m
def exe(m, g, v):
    unch = [n for n in set(v).intersection(m.ctrl).
        difference(g.ctrl)] # the variables g leaves
        unchanged
    updt = [a.var for a in g.actions if a.b.eval(v)] # the
        variables g actually updates
```

```
    return sorted(updt+unch)

# returns the valuation obtained by executing a joint
    guarded command
def jointExe(modules, J, v):
    N = len(modules)
    w = []
    for i in range(N):
        w += exe(modules[i], J[i], v)
    return sorted(w)

# returns the list of commands enabled for module m at
    valutation v
def enabled(m, v):
    if [(m, g) for g in m.update if g.guard.eval(v)]:
        return [g for g in m.update if g.guard.eval(v)]
    else:
        return [Command(T, [])]

# returns the list of commands enabled for module m at
    initiation
def enabled_init(m):
    if [(m, g) for g in m.init if g.guard.eval([])]:
        return [g for g in m.init if g.guard.eval([])]
    else:
        return [Command(T, [])]

# returns the cartesian product of enabled commands for a
    list of modules, at valuation v
def jointEnabled(modules, v):
    return product(*[enabled(m, v) for m in modules])

# returns the cartesian product of enabled commands for a
    list of modules, at initiation
def jointEnabled_init(modules):
    return product(*[enabled_init(m) for m in modules])


# translates the abstract syntax tree of a formula into a
    PL instance
def ast2pl(ast):
    if len(ast) == 1 and ast[0] == "F":
        return F
    elif len(ast) == 1 and ast[0] == "T":
        return T
    elif len(ast) == 1:
```

```
        assert(ast[0][0] == 'x')
        return Var(int(ast[0][1:]))
    elif ast[0] == "!":
        return Not(ast2pl(ast[1]))
    elif ast[0] == "or":
        return Or(ast2pl(ast[1]), ast2pl(ast[2]))
    elif ast[0] == "and":
        return And(ast2pl(ast[1]), ast2pl(ast[2]))

# These functions parse the input file into RM instances.

# parses a propositional formula
def parsePL(s):
    ast = CTL.parse(s)
    return ast2pl(ast)

# parses a list of actions
def parseActions(xs):
    l = list()
    for x in xs.split(', '):
        assert(len(x.split(' := ')) == 2)
        var = int(x.split(' := ')[0][1:])
        b = parsePL(x.split(' := ')[1])
        l.append(Action(var, b))
    return l

# parses a guarded command
def parseCommand(g):
    assert(len(g.split(' -> ')) == 2)

    guard = parsePL(g.split(' -> ')[0])
    actions = parseActions(g.split(' -> ')[1])

    return Command(guard, actions)

# parses a reactive module
def parseRM(m):
    return RM(m['ctrl'], [parseCommand(g) for g in m['init'
        ]],
        [parseCommand(g) for g in m['update']])
```

pl.py

```
# This module defines a class for Propositional Logic (PL)
    formulae
# The set of propositional letters is written x0, x1, ...
    and variables are stored as ints.
# A valuation is simply a list of ints: variable x_n
    evaluates to True iff n is in the list
# A PL instance has a method eval which, given a valuation
    list, returns True iff the valuation satisfies the
    formula

class PL:
    def __repr__(self):
        return repr(self)

class T(PL):
    def __init__(self):
        self.type = "T"

    def __repr__(self):
        return "T"

    def eval(self, v):
        return True

T = T()

class F(PL):
    def __init__(self):
        self.type = "F"

    def __repr__(self):
        return "F"

    def eval(self, v):
        return False

F = F()

class Var(PL):
    def __init__(self, n):
        self.n = n
        self.type = "Var"

    def __repr__(self):
```

```
        return "x"+str(self.n)

    def eval(self, v):
        return self.n in v

class Not(PL):
    def __init__(self, p):
        self.p = p
        self.type = "Not"

    def __repr__(self):
        return "!"+repr(self.p)

    def eval(self, v):
        return not self.p.eval(v)

class Or(PL):
    def __init__(self, p, q):
        self.p = p
        self.q = q
        self.type = "Or"

    def __repr__(self):
        return "(or "+repr(self.p)+" "+repr(self.q)+")"

    def eval(self, v):
        return self.p.eval(v) or self.q.eval(v)

class And(PL):
    def __init__(self, p, q):
        self.p = p
        self.q = q
        self.type = "And"

    def __repr__(self):
        return "(and "+repr(self.p)+" "+repr(self.q)+")"

    def eval(self, v):
        return self.p.eval(v) and self.q.eval(v)
```

thctl.py

```
# This module implements the translation from a list of
    modules to the corresponding CTL theory.
# The Kripke structures that satisfy the theory are exactly
     those whose tree unfoldings correspond to the arena's.

from pl import *
from rm import *

import subprocess


# characters reserved by the CTLSAT parser, not to be used
    as atoms
notAtoms = ['T','B','S','N','$',')','(','>','U','^','v','A'
    ,'E','C','D','H','I','J','K','~']

# characters to be used as atoms
atoms = [chr(i) for i in range(48, 127) if chr(i) not in
    notAtoms]

# maps variables to ascii characters to be parsed by CTLSAT
def var2chr(n, p='mu'):
    assert(n <= len(atoms)) # asserts we haven't ran out of
        characters for atoms

    return atoms[n]

# translates PL instances into strings to be input by
    CTLSAT
# used for guards and for Boolean values in update commands
def pl2ctl(f):
    if f.type == "T":
        return "T"
    elif f.type == "F":
        return "(~T)"
    elif f.type == "Var":
        return var2chr(f.n)
    elif f.type == "Not":
        return "(~"+pl2ctl(f.p)+")"
    elif f.type == "Or":
        return "("+pl2ctl(f.p)+"v"+pl2ctl(f.q)+")"
    elif f.type == "And":
        return "("+pl2ctl(f.p)+"^"+pl2ctl(f.q)+")"
```

```
# translates the abstract syntax tree of a formula into a
    string to be input by CTLSAT
# used for goals
def ast2ctl(ast):
    if len(ast) == 1 and ast[0] == "true":
        return "T"
    elif len(ast) == 1 and ast[0] == "false":
        return "(~T)"
    elif len(ast) == 1:
        return var2chr(int(ast[0][1:]))
    elif ast[0] == "!":
        return "(~"+ast2ctl(ast[1])+")"
    elif ast[0] == "or":
        return "("+ast2ctl(ast[1])+"v"+ast2ctl(ast[2])+")"
    elif ast[0] == "and":
        return "("+ast2ctl(ast[1])+"^"+ast2ctl(ast[2])+")"
    elif ast[0] == "AX":
        return "(AX"+ast2ctl(ast[1])+")"
    elif ast[0] == "EX":
        return "(EX"+ast2ctl(ast[1])+")"
    elif ast[0] == "EF":
        return "(EF"+ast2ctl(ast[1])+")"
    elif ast[0] == "AF":
        return "(AF"+ast2ctl(ast[1])+")"
    elif ast[0] == "AG":
        return "(AG"+ast2ctl(ast[1])+")"
    elif ast[0] == "EG":
        return "(EG"+ast2ctl(ast[1])+")"
    elif ast[0] == "EU":
        return "(E("+ast2ctl(ast[1])+"U"+ast2ctl(ast[2])+"))"
    elif ast[0] == "AU":
        return "(A("+ast2ctl(ast[1])+"U"+ast2ctl(ast[2])+"))"


# builds the string for the XOR of a set of formulae
def XOR(fs):
    s = ""
    for i in range(len(fs)):
        s += "v("+fs[i]
        for j in range(len(fs)):
            if i != j:
                s += "^~("+fs[j]+")"
        s += ")"
    return "("+s[1:]+")"

# builds the string for the AND of a set of formulae
```

```
def AND(fs):
    if not fs:
        return "T"
    else:
        s = ""
        for f in fs:
            s += "^"+f
        return s[1:]

# builds the string for the IFF of a set of formulae
def IFF(fs):
    s = ""
    for i in range(len(fs)-1):
        s += "^("+fs[i]+"->"+fs[i+1]+")"
        s += "^("+fs[i+1]+"->"+fs[i]+")"
    return s[1:]

# predicate for: "a' := b"
def ASSIGN(a, b):
    if b == "T":
        return "(AX"+a+")"
    elif b == "(~T)":
        return "(AX~"+a+")"
    else:
        return "("+b+"->(AX"+a+"))^(~"+b+"->(AX~"+a+"))"

# predicate for: "the variables in vs remain unchanged at
    the next round"
def UNCH(vs):
    return AND([ASSIGN(var2chr(v), var2chr(v)) for v in vs])

# predicate for: "init command g is executed by a module
    controlling variables mctrl"
def INITcmd(mctrl, g):
    xs = list()

    if [1 for a in g.actions if a.b.eval([])]: # "x' := T"
        xs.append(AND([var2chr(a.var) for a in g.actions if a
            .b.eval([])]))

    if [1 for a in g.actions if not a.b.eval([])]: # "x' :=
        F"
        xs.append(AND(["~"+var2chr(a.var) for a in g.actions
            if not a.b.eval([])]))

    if [1 for v in mctrl if v not in g.ctrl]: # variables
```

```
        not mentioned in g
        xs.append(AND(["~"+var2chr(v) for v in mctrl if v not
            in g.ctrl]))

    return AND(xs)

# predicate for: "module m executes exactly one init
    command"
def INIT(m):
    return XOR([INITcmd(m.ctrl, g) for g in m.init])

# predicate for: "update command g is executed by a module
    controlling variables mctrl"
def UPDATEcmd(mctrl, g):
    xs = list()

    if g.guard.type != "T": # no need for the guard if T
        xs.append(pl2ctl(g.guard))

    if [1 for a in g.actions]: # "x' := b"
        xs.append(AND([ASSIGN(var2chr(a.var), pl2ctl(a.b))
            for a in g.actions]))

    if [1 for v in mctrl if v not in g.ctrl]: # variables
        not mentioned in g
        xs.append(UNCH([v for v in mctrl if v not in g.ctrl])
            )

    return AND(xs)

# predicate for: "module m executes exactly one update
    command"
def UPDATE(m):
    if True: #[1 for g in m.update if g.guard.type == "T"]:
        # no need for the null-guarded command if there is
            always an enabled command
        return XOR([UPDATEcmd(m.ctrl, g) for g in m.update])
    else:
        nullCommand = "("+UNCH(m.ctrl)+"^"+AND(["~"+pl2ctl(g.
            guard) for g in m.update])+")"
        return nullCommand+"v"+XOR([UPDATEcmd(m.ctrl, g) for
            g in m.update])

# CTL theory for a list of modules
def Th(modules):
    # predicate for: "every module executes exactly one init
```

```
        command"
    INITS = AND([INIT(m) for m in modules])

    # predicate for: "every module executes exactly one
        update command"
    UPDATES = AND([UPDATE(m) for m in modules])

    return INITS+"^AG("+UPDATES+")"

# calls CTLSAT as a subprocess and returns True iff the
    formula is satisfiable
def SAT(f, v=False):
    if v: print; print f

    command = subprocess.Popen(['CTLSAT-master/ctl-sat', f],
        stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    out, err = command.communicate()

    return out.split('\n')[-2]== "Input formula is
        satisfable!"
```

arena2kripke.py

```
# This module implements the translation from a list of
    modules to a Kripke structure.
# The unfoldings of the Kripke structure are in strong
    correspondance with the outcome of the game.
# Nodes of the structure are valuations of the variables
    controlled by the modules.
# There is an edge between nodes u and v whenever there is
    a joint guarded command J that is enabled at u, and exec
    (J, u) = v

from rm import *

from networkx import *
from mrwaffles.checker import *
from mrwaffles.ctl import *


import pickle


# Nodes need to be labeled with hashable types, so we
    convert valuation lists into ints
#  such that the i-th digit of lab(v) is 1 iff v maps
    variable i to true.

# maps valuations to int labels
def lab(v):
    n = 0
    for x in v:
        n += 2**x
    return n

# maps int labels to valuations
def lab2val(n):
    x = 0
    v = []
    while n > 0:
        if n % 2 == 1:
            v.append(x)
        n = n/2
        x += 1
    return sorted(v)

# translates a list of modules into a mrwaffles
```

```
    PredicatedGraph
def Arena2Kripke(modules):
    A = PredicatedGraph()
    S0 = set()
    S = set()

    for J in jointEnabled_init(modules):
        v = jointExe(modules, J, [])
        vlab = lab(v)

        if vlab not in S:
            S0.add(vlab)
            S.add(vlab)
            A.add_node(vlab)

            for x in v:
                A.add_predicate(vlab, 'x'+str(x))

    X = set()
    while X != S:
        X = copy.copy(S)
        for vlab in X:
            v = lab2val(vlab)

            for J in jointEnabled(modules, v):
                w = jointExe(modules, J, v)
                wlab = lab(w)

                if not wlab in S:
                    S.add(wlab)
                    A.add_node(wlab)
                    for x in w:
                        A.add_predicate(wlab, 'x'+str(x))

    for vlab in S:
        for wlab in S:
            v = lab2val(vlab)
            w = lab2val(wlab)
            for J in jointEnabled(modules, v):
                if w == jointExe(modules, J, v):
                    A.add_edge(vlab, wlab)
                    break

    return (A, S0)

# return true iff Kripke structure A with starting states
```

```
    S0 satisfies the formula goal
def entails(A, S0, goal):
    states = A.check(goal)

    for s in S0:
        if not s in states:
            return False
    return True
```

References

[1] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani and S. Taşiran. Mocha: Modularity in Model Checking. In *CAV*. Springer, 2009.

[2] R. Alur and T.A. Henzinger and O. Kupferman. Alternating-time temporal logic. In *ACM*, 2002.

[3] K. Chatterjee, T.A. Henzinger and N. Piterman. Strategy logic In *Inf. Comput.*, 2010.

[4] S.F. Cheng, D.M. Reeves, Y. Vorobeychik, M.P. Wellman. Notes on Equilibria in Symmetric Games. In *AAMAS-04 Workshop on Game Theory and Decision Theory*, 2004.

[5] P. Clairambault, J. Gutierrez and G. Winskel. Imperfect Information in Logic and Concurrent Games. In *Computation, Logic, Games, and Quantum Foundations*, 2013.

[6] E.M. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency.* Springer, 1988.

[7] J. Conway. *On Numbers and Games*. Academic Press Inc., 1976.

[8] E.A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[9] D. Fisman, O. Kupferman and Y. Lustig. Rational Synthesis. In *TACAS*. Springer, 2010.

[10] J. Gutierrez, P. Harrenstein, and M. Wooldridge. On the Complexity of Equilibria in Reactive Modules Games. Available at: *http://www.cs.ox.ac.uk/people/julian.gutierrez/web/RMGpaper.pdf* Unpublished, 2015.

[11] J. Gutierrez, P. Harrenstein, and M. Wooldridge. Iterated Boolean games. In *IJCAI*. IJCAI/AAAI Press, 2013.

[12] W. van der Hoek and M. Wooldridge. Towards a logic of rational agency. IN *Logic Journal of the IGPL*, 2003.

[13] M. Kwiatkowska, G. Norman and D. Parker. PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. In *ACM SIGMETRICS Performance Evaluation Review*, 2009.

[14] O. Morgenstern and J. von Neumann. *Theory of Games and Economic Behavior (60th-Anniversary Edition)*. Princeton University Press, 2007.

[15] J.F. Nash, Jr. *Non-Cooperative Games*. PhD thesis, Princeton University, Department of Mathematics, 1950.

[16] A. Procaccia Is Game Theory (Artificially) Intelligent? In *Turing's Invisible Hand*, 2011. Available at: *https://agtb.wordpress.com/2011/01/18/is-game-theory-artificially-intelligent/*

[17] N. Prezza. CTLSAT. Available at: *https://github.com/nicolaprezza/CTLSAT*, 2015.

[18] D. Reynaud. Mr. Waffles. Available at: *http://mrwaffles.gforge.inria.fr*, 2009.

[19] A. Turing. Computing Machinery and Intelligence. In *Mind*, 1950.

[20] M. Wooldridge, T. Ågotnes, P.E. Dunne and W. van der Hoek. Logic for Automated Mechanism Design - A Progress Report In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, 2007.