# A Tool for the Automated Verification of Nash Equilibria in Concurrent Games

Alexis Toumi, Julian Gutierrez(✉), and Michael Wooldridge

Department of Computer Science, University of Oxford, Oxford, UK
Julian.Gutierrez@cs.ox.ac.uk

**Abstract.** Reactive Modules is a high-level specification language for concurrent and multi-agent systems, used in a number of practical model checking tools. Reactive Modules Games is a game-theoretic extension of Reactive Modules, in which concurrent agents in the system are assumed to act strategically in an attempt to satisfy a temporal logic formula representing their individual goal. The basic analytical concept for Reactive Modules Games is Nash equilibrium. In this paper, we describe a tool through which we can automatically verify Nash equilibrium strategies for Reactive Modules Games. Our tool takes as input a system, specified in the Reactive Modules language, a representation of players' goals (expressed as CTL formulae), and a representation of players strategies; it then checks whether these strategies form a Nash equilibrium of the Reactive Modules Game passed as input. The tool makes extensive use of conventional temporal logic satisfiability and model checking techniques. We first give an overview of the theory underpinning the tool, briefly describe its structure and implementation, and conclude by presenting a worked example analysed using the tool.

## 1 Introduction

Model checking is the best-known and most successful technique for automated formal verification, and is focussed on the problem of checking whether a (computer) system $S$ satisfies a property $\varphi$, where typically $\varphi$ is represented as a temporal logic formula. Model checking has proved to be a very successful technique for systems where $S$ is a complete and monolithic description of the state space of the system. In this case, $S$ is usually called a *closed* system. However, in many situations, especially when dealing with concurrent and distributed multi-agent systems, $S$ can be better represented as a collection of local and inter-dependent processes. In this modelling framework, it is common to understand such processes as *modules*, that is, as being *open* rather than closed systems, in which the behaviour of each process/module may depend on the behaviour of other processes, which constitute its environment, cf., [2,12].

We are interested in the verification of concurrent and multi-agent systems where (computer) processes are modelled as open systems. In particular, we are interested in systems modelled using a game-theoretic approach. In this setting, a system is modelled as a *game*, system components are modelled as *players*


M. Leucker et al. (Eds.): ICTAC 2015, LNCS 9399, pp. 583–594, 2015.
DOI: 10.1007/978-3-319-25150-9_34

(each choosing and then following a given *strategy*), possible computation runs are the *plays* of the game, and the desired or expected behaviour of the system is specified with the *goals* that the players of the game wish to see satisfied. In many cases, for instance when considering reactive systems, such goals can be naturally expressed using temporal logic formulae.

However, because now one is following a game-theoretic approach, it is only natural to ask whether the system has a stable behaviour from a game-theoretic point of view, that is, whether the strategies used by the players modelling the system are in equilibrium [15]. Then, in this case, we talk about *equilibrium checking* rather than model checking. In fact, model checking is a simpler instance of equilibrium checking where either players are forced to cooperate or the whole system is modelled as a one-player game. However, in general, these may not be the best representations of the system.

A way to model the kind of systems just described (open systems) is using the Reactive Modules Language (RML [2]). This is a high-level specification language for reactive, concurrent, and multi-agent systems, which is used in model checking tools such as MOCHA [1] and Prism [13]. However, RML is used to specify general open systems rather than concurrent games. Recently [11], a subset of RML, called the Simple Reactive Modules Language (SRML [19]) was given a game-theoretic interpretation, which provides a game semantics for reactive and concurrent systems written in SRML, and which can be used to perform an equilibrium analysis of open systems modelled as SRML specifications. Indeed, with SRML, one can analyse systems using a language that is much closer to real-world programming and system modelling languages.

In this paper, we present a tool for the automated verification of Nash equilibria in concurrent and reactive systems modelled as concurrent games succinctly represented using the SRML specification language. More specifically, we develop a Python implementation of the above theory of games that, in particular, can be used to solve the *equilibrium checking* problem for this kind of concurrent games/systems. Since the tool, which we call EAGLE ("**E**quilibrium **A**nalyser for **G**ame-**L**ike **E**nvironments"), can be used to automatically check whether a set of strategies forms a *Nash Equilibrium* in a given game-like concurrent system, its analytical power goes beyond model checking.

**Related Work.** Reactive Modules [2] is used as a specification language in verification tools such as MOCHA [1] and Prism [13]. In each case, open systems modelled as concurrent games can also be specified. However, these tools do not have explicit support for equilibrium analysis. Instead, it is model checking with respect to logics such as PCTL and ATL that these tools allow. MCMAS [14] is another tool for the specification and verification of open systems, modelled as multi-agent systems. In MCMAS, systems are described using the Interpreted Systems Programming Language and properties are described using ATL$^*$ and strategy logic—see [5]. Similar to MOCHA and Prism, in MCMAS the analysis of systems focuses on the model checking problem for the logics just mentioned. Because strategy logic can express the existence of Nash equilibria in a concurrent and multi-agent game, in principle, it is possible to analyse some equilibrium properties of MCMAS systems. However, this has to be manually crafted.

Closer to EAGLE is PRALINE [4], a tool for computing Nash equilibria in concurrent games played on graphs. Whereas PRALINE focuses on the synthesis problem (constructing strategies in equilibrium), EAGLE focuses on the verification problem (checking that a given profile of strategies is in equilibrium). There are many other tools available online which either use game techniques for design and verification or allow the analysis of winning strategies in games. For instance, see [3,6,9] for a few references. However, as just said, these tools focus on the study of winning strategies in such games rather than in the equilibrium analysis of these systems/games.

## 2 Preliminaries

*Logic.* In this paper we will be dealing with logics that extend classical propositional logic. Thus, these logics are based on a finite set $\Phi$ of Boolean variables. A *valuation* for propositional logic is a set $v \subseteq \Phi$, with the intended interpretation that $p \in v$ means that $p$ is true under valuation $v$, while $p \notin v$ means that $p$ is false under $v$. For formulae $\varphi$ we write $v \models \varphi$ to mean that $\varphi$ is satisfied by $v$. Let $V(\Phi) = 2^{\Phi}$ be the set of all valuations for variables $\Phi$; where $\Phi$ is clear, we omit reference to it and simply write $V$.

*Kripke Structures.* We use *Kripke structures* to model the dynamics of our systems. A Kripke structure $K$ over $\Phi$ is given by $K = (S, S^0, R, \pi)$, where $S = \{s_0, \ldots\}$ is a finite non-empty set of *states*, $R \subseteq S \times S$ is a total *transition relation* on $S$, $S^0 \subseteq S$ is the set of *initial states*, and $\pi : S \to V$ is a valuation function, assigning a valuation $\pi(s)$ to every $s \in S$. Where $K = (S, S^0, R, \pi)$ is a Kripke structure over $\Phi$, and $\Psi \subseteq \Phi$, then we denote the *restriction of $K$ to $\Psi$* by $K|_\Psi$, where $K|_\Psi = (S, S^0, R, \pi|_\Psi)$ is the same as $K$ except that the valuation function $\pi|_\Psi$ is defined as follows: $\pi|_\Psi(s) = \pi(s) \cap \Psi$.

*Runs.* A *run of $K$* is a sequence $\rho = s_0, s_1, s_2, \ldots$ where for all $t \in \mathbb{N}$ we have $(s_t, s_{t+1}) \in R$. Using square brackets around parameters referring to time points, we let $\rho[t]$ denote the state assigned to time point $t$ by run $\rho$. We say $\rho$ is an $s$-run if $\rho[0] = s$. A run $\rho$ of $K$ where $\rho[0] \in S^0$ is referred to as 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 $K$. Notice that a run $\rho \in runs(K)$ induces an infinite sequence $\boldsymbol{\rho} \in V^\omega$ of propositional valuations, viz., $\boldsymbol{\rho} = \pi(\rho[0]), \pi(\rho[1]), \pi(\rho[2]), \ldots$. The set of these sequences, we denote by $\mathbf{runs}(K)$. Given $\Psi \subseteq \Phi$ and a run $\boldsymbol{\rho}\colon \mathbb{N} \to V(\Phi)$, we denote the restriction of $\boldsymbol{\rho}$ to $\Psi$ by $\boldsymbol{\rho}|_\Psi$, that is, $\boldsymbol{\rho}|_\Psi[t] = \boldsymbol{\rho}[t] \cap \Psi$ for each $t \in \mathbb{N}$. We can extend the notation for restriction of runs to sets of runs. In particular, we write $\mathbf{runs}(K)|_\Psi$ for the set $\{\boldsymbol{\rho}|_\Psi : \boldsymbol{\rho} \in \mathbf{runs}(K)\}$.

*Trees.* By a *tree* we here understand a non-empty set $T \subseteq \mathbb{N}_0^*$, such that $(i)$ $T$ is closed under prefixes, *i.e.*, for every $u \in T$, also $(u) \subseteq T$, and $(ii)$ $u \in T$ implies $ux \in T$ for some $x \in \mathbb{N}_0$. For $s \in S$, a *state-tree* for a Kripke structure $K = (S, S^0, R, \pi)$ is a function $\kappa\colon T \to S$, where $T \subseteq \mathbb{N}_0^*$ is a tree, $\kappa(\epsilon) \in S^0$,

and, for every $u \in \mathbb{N}_0^*$ and $x, y \in \mathbb{N}_0$ such that $ux, uy \in T$, $(i)$ $\kappa(u)\ R\ \kappa(ux)$, and $(ii)$ $\kappa(ux) = \kappa(uy)$ implies $x = y$. By $trees(K)$ we denote the state-trees for the Kripke structure $K$. By a *computation tree* we understand a function $\boldsymbol{\kappa}\colon T \to V(\Phi)$, where $T$ is a tree. For $\Psi \subseteq \Phi$ we write $\boldsymbol{\kappa}|_\Psi$ for the restriction of $\boldsymbol{\kappa}$ to $\Psi$, *i.e.*, for every $u \in T$, $\boldsymbol{\kappa}|_\Psi(u) = \boldsymbol{\kappa}(u) \cap \Psi$. Notice that every state-tree $\kappa\colon T \to S$ induces a computation tree $\boldsymbol{\kappa}\colon T \to V(\Phi)$ such that for every $u \in T$ we have that $\boldsymbol{\kappa}[u] = \pi(\kappa(u))$. In such a case $\boldsymbol{\kappa}$ is said to be a computation tree for $K$. The set of computation trees for $K$ we denote by $\mathbf{trees}(K)$. We can extend the notation for restrictions of computation trees to sets of computation trees as done for runs, that is, we write $\mathbf{trees}(K)|_\Psi$ for the set $\{\boldsymbol{\kappa}|_\Psi : \boldsymbol{\kappa} \in \mathbf{trees}(K)\}$.

## 3 Reactive Modules Games

*Reactive Modules.* The objects used to define agents in SRML are known as *modules*. An SRML module consists of: *(i)* an *interface*, which defines the name of the module and lists the Boolean variables under the *control* of the module; and *(ii)* a number of *guarded commands*, which define the choices available to the module at every state.

Guarded commands are of two kinds: those used for *initialising* the variables under the module's control (**init** guarded commands), and those for *updating* these variables subsequently (**update** guarded commands). A guarded command has two parts: a condition part (the "guard") and an action part, which defines how to update the value of (some of) the variables under the control of a module. The intuitive reading of a guarded command $\varphi \rightsquigarrow \alpha$ is "if the condition $\varphi$ is satisfied, then *one of the choices available to the module is to execute the action* $\alpha$". We note that the truth of the guard $\varphi$ does not mean that $\alpha$ *will* be executed: only that it is *enabled* for execution—it *may be chosen*.

Formally, a guarded command $g$ over a set of Boolean variables $\Phi$ is an expression

$$\varphi \rightsquigarrow x_1' := \psi_1; \cdots ; x_k' := \psi_k$$

where $\varphi$ (the guard) is a propositional formula over $\Phi$, each $x_i$ is a member of $\Phi$ and each $\psi_i$ is a propositional logic formula over $\Phi$. Let $guard(g)$ denote the guard of $g$. Thus, in the above rule, $guard(g) = \varphi$. We require that no variable appears on the left hand side of two assignment statements in the same guarded command. We say that $x_1, \ldots, x_k$ are the *controlled variables* of $g$, and denote this set by $ctr(g)$. If no guarded command of a module is enabled, the values of all variables in $ctr(g)$ are left unchanged.

Formally, an SRML module, $m_i$, is defined as a triple $m_i = (\Phi_i, I_i, U_i)$, where: $\Phi_i \subseteq \Phi$ is the (finite) set of variables controlled by $m_i$; $I_i$ is a (finite) set of *initialisation* guarded commands, such that for all $g \in I_i$, we have $ctr(g) \subseteq \Phi_i$; and $U_i$ is a (finite) set of *update* guarded commands, such that for all $g \in U_i$, we have $ctr(g) \subseteq \Phi_i$.

Moreover, an SRML *arena*, $A$, is defined to be an $(n+2)$-tuple

$$A = (N, \Phi, m_1, \ldots, m_n)$$

where $N = \{1, \ldots, n\}$ is a set of agents, $\Phi$ is a set of Boolean variables, and for each $i \in N$, $m_i = (\Phi_i, I_i, U_i)$ is an SRML module over $\Phi$ that defines the choices available to agent $i$. We require that $\{\Phi_1, \ldots, \Phi_n\}$ forms a partition of $\Phi$ (so every variable in $\Phi$ is controlled by some agent, and no variable is controlled by more than one agent).

The behaviour of an SRML arena is obtained by executing guarded commands, one for each module, in a synchronous and concurrent way. The execution of an SRML arena proceeds in rounds, where in each round every module $m_i = (\Phi_i, I_i, U_i)$ produces a valuation $v_i$ for the variables in $\Phi_i$ on the basis of a current valuation $v$. For each SRML arena $A$, the execution of guarded commands induces a unique Kripke structure $K_A$, which formally defines the semantics of $A$. Based on $K_A$, one can define the sets of runs and computation trees allowed in $A$, namely, those associated with the Kripke structure $K$; we write $\mathbf{runs}(A)$ and $\mathbf{trees}(A)$ for such sets. Indeed, one can show that for every $A$ there is a $K_A$ such that $\mathbf{runs}(A) = \mathbf{runs}(K_A)|_\Phi$ and $\mathbf{trees}(A) = \mathbf{trees}(K_A)|_\Phi$, that is, with the same runs and computation trees when restricted to $\Phi$. Likewise, for every $K$ there is an SRML module whose runs and computation trees are those of $K$. In this paper, we provide, amongst others, a Python implementation of all these constructions.

*Games.* The model of games we consider has two components. The first component is an *arena*: this defines the players, some variables they control, and the choices available to them in every game state. Preferences are specified by the second component of the game: every player $i$ is associated with a *goal* $\gamma_i$, which will be a logic formula. The idea, as in several models of strategic behaviour, is that players desire to see their goal satisfied by the outcome of the game. Formally, a game is given by a structure:

$$G = (A, \gamma_1 \ldots, \gamma_n)$$

where $A = (N, \Phi, m_1, \ldots, m_n)$ is an arena with player set $N$, Boolean variable set $\Phi$, and $m_i$ an SRML module defining the choices available to each player $i$; moreover, for each $i \in N$, the temporal logic formula $\gamma_i$ represents the *goal* that $i$ aims to satisfy.[1] Games are played by each player $i$ selecting a *strategy* $\sigma_i$ that will define how to make choices over time. Given an SRML arena $A = (N, \Phi, m_1, \ldots, m_n)$, a *strategy* for module $m_i = (\Phi_i, I_i, U_i)$ is a structure $\sigma_i = (Q_i, q_i^0, \delta_i, \tau_i)$, where $Q_i$ is a finite and non-empty set of *states*, $q_i^0 \in Q_i$ is the *initial* state, $\delta_i \colon Q_i \times V_{-i} \to 2^{Q_i} \setminus \{\emptyset\}$ is a *transition function*, and $\tau_i \colon Q_i \to V_i$ is an *output function*. Note that not all strategies for a module may comply with that module's specification. For instance, if the only guarded update command of a module $m_i$ has the form $\top \rightsquigarrow x' := \bot$, then a strategy for $m_i$ should not prescribe $m_i$ to set $x$ to true under any contingency. Strategies that comply with

[1] Goals can be given by any logic with a Kripke structure semantics. Although we will consider CTL goals here, due to generality, at this point all definitions will be made leaving this open. Indeed, one could extend our implementation to SRML games with CTL$^*$ or $\mu$-calculus goals.

the module's specification are called consistent. Let $\Sigma_i$ be the set of consistent strategies for $m_i$. A strategy $\sigma_i$ can be represented by an SRML module (of polynomial size in $|\sigma_i|$) with variable set $\Phi_i \cup Q_i$. We write $m_{\sigma_i}$ for such a (strategy) module specification.

Once every player $i$ has selected a strategy $\sigma_i$, a *strategy profile* $\vec{\sigma} = (\sigma_1, \ldots, \sigma_n)$ results and the game has an *outcome*, which we will denote by $[\![\vec{\sigma}]\!]$. The outcome $[\![\vec{\sigma}]\!]$ of a game with SRML arena $A = (N, \Phi, m_1, \ldots, m_n)$ is defined to be the Kripke structure associated with the SRML arena $A_{\vec{\sigma}} = (N, \Phi \cup \bigcup_{i \in N} Q_i, m_{\sigma_1}, \ldots, m_{\sigma_n})$ restricted to valuations with respect to $\Phi$, that is, the Kripke structure $K_{A_{\vec{\sigma}}}|_{\Phi}$. The outcome of a game will determine whether or not each player's goal is or is not satisfied. Because outcomes are Kripke structures, in general, goals can be given by any logic with a well defined Kripke structure semantics. Assuming the existence of such a satisfaction relation, which we denote by "$\models$", we can say that a goal $\gamma_i$ is satisfied by an outcome $[\![\vec{\sigma}]\!]$ if and only if $[\![\vec{\sigma}]\!] \models \gamma_i$; in order to simplify notations, we may simply write $\vec{\sigma} \models \gamma_i$.

We are now in a position to define a preference relation $\succsim_i$ over outcomes for each player $i$ with goal $\gamma_i$. For strategy profiles $\vec{\sigma}$ and $\vec{\sigma}'$, we say that

$$\vec{\sigma} \succsim_i \vec{\sigma}' \text{ if and only if } \vec{\sigma}' \models \gamma_i \text{ implies } \vec{\sigma} \models \gamma_i.$$

On this basis, we can also define the standard solution concept of Nash equilibrium [15]: given a game $G = (A, \gamma_1, \ldots, \gamma_n)$, a strategy profile $\vec{\sigma}$ is said to be a *Nash equilibrium* of $G$ if for all players $i$ and all strategies $\sigma_i'$ in the game, we have

$$\vec{\sigma} \succsim_i (\vec{\sigma}_{-i}, \sigma_i'),$$

where $(\vec{\sigma}_{-i}, \sigma_i')$ denotes the strategy profile $(\sigma_1, \ldots, \sigma_{i-1}, \sigma_i', \sigma_{i+1}, \ldots, \sigma_n)$. Hereafter, let $NE(G)$ be the set of (pure strategy) Nash equilibria of game $G$.

## 4 Reactive Modules Games in Python

Our main contribution is EAGLE, *a Python implementation of the theory of games* described in the previous sections. In particular, EAGLE allows a simple high-level Python description of games specified in SRML, where players are assumed to have branching-time (CTL) goals and strategies can be described as SRML modules. More importantly, EAGLE allows the automated verification of solutions of such games, that is, checking whether a particular profile of strategies is or is not a Nash equilibrium of a given RM game—a problem called *equilibrium checking*. From a systems analysis point of view, this is the game-theoretic equivalent to the *model checking* problem in formal verification. A short description of our verification tool is given next.

Our tool expects as input an RM game $G = (A = (N, \Phi, m_0, ..., m_n), (\gamma_i)_{i \in N})$ and a strategy profile $\vec{\sigma}$. Because strategies are modelled as finite state machines with output (which are known as transducers), they can easily be described, uniformly, using SRML. Goals, on the other hand, are written using the syntax for CTL formulae in [7]. For ease of use, a simple command-line interface can be used to input text files with the specification of games. An concrete example will

be given later, but all implementation details can be found in [18]. Moreover, EAGLE implements an algorithm—which uses two external libraries for CTL satisfiability and model checking—that automatically solves these multi-player games, that is, their (Nash) equilibrium problem.

More precisely, on input $(G, \vec{\sigma})$, the tool outputs **True** if and only if $\vec{\sigma} \in NE(G)$. We have also implemented, using the command-line interface, a "verbose" mode in which a detailed account of the running process of the algorithm is given. For instance, apart from checking solutions of a given game, the tool reports whether or not players get their goal achieved, and in the case they do not, whether they could benefit from changing the strategy they are currently using. We should note that because in a Nash equilibrium strategy profile no player can benefit from unilaterally changing its strategy, it is the case that if the tool reports that $\vec{\sigma} \notin NE(G)$, then there is some player who does not get its goal achieved, but can change to a different strategy that achieves its goal. On the contrary, if the tool reports that $\vec{\sigma} \in NE(G)$, then no player can benefit from changing its strategy, in particular, those who do not get their goal achieved.

Throughout, we made the following assumptions, which define what a correct input is. In some cases, the assumptions are about the games themselves (1 & 2), and in other cases about the input files (3). In particular, we have made the following assumptions:

1. That the modules, both for the arena and for the strategy profile, respect the specification of SRML. In particular, we require: (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, $\psi$ is a Boolean constant, $\top$ or $\bot$; (d) that for every module $m_i = (\Phi_i, I_i, U_i)$, both $I_i$ and $U_i$ are sets instead of bags, i.e. that they contain only pairwise distinct elements; (e) that for every 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 strategy profile is consistent with the arena, as required by the game model.
3. That the input strings for goals are syntactically correct CTL formulae, in particular that they respect the alternation between path quantifiers and tense operators.

To make this concrete, we will, later and in the next section, present some examples.

*CTL Satisfiability and Model Checking.* In order to solve the equilibrium problem for Reactive Modules games we used a CTL variant of the algorithms first introduced in [10] to check whether a strategy profile is or is not a Nash equilibrium. The technique developed in [10] relies on the existence of two oracles, one for model checking and one for satisfiability of the temporal logic at hand. In the case of this paper, such oracles are for CTL, and can be obtained using any "off-the-shelf" open source external libraries for CTL satisfiability (Ctl Sat) and CTL model checking (Ctl MC). Specifically, we decided to use the Python CTL model checker Mr.Waffles [17] and the CTL satisfiability checker in [16], both open source libraries available online.

For Ctl MC, the Mr.Waffles library implements Kripke structures with a class `PredicatedGraph` which extends the `networkx` library for finite 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 a Kripke structure satisfies a CTL formula amounts to checking that all the initial states are in this list. For Ctl Sat, we use a command-line interface to access an external program that inputs CTL formulae as strings (in infix notation), which is wrapped using a Python subprocess instance.

*Concrete Data Structures.* We 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 $\bot$. Also, 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.[2]

*Input Format.* As expected we use Python files, which we then parse 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. More specifically, we represent modules as Python dictionaries, following the same structure as the named tuples for modules described before. 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`. At this point it is worth noting that using our Python assistant any finite-state strategy can be represented, including non-deterministic ones, by extending the set of controlled variables to represent strategy states without affecting the outcome of the game (of course, as long as the strategy is consistent with its module).

*System Architecture.* Our system has five Python modules, as follows: 1. A module that implements the command-line interface and the main algorithm; it also implements the verbose mode and prints some running time measurements.

[2] EAGLE is being improved and updated frequently. The implementation details in this paper constitute the main design decisions at the moment of submission to ICTAC (in June 2015).

2. A module that implements the propositional logic class. 3. A module that implements the concrete data structures described before, as well as the parsing of input modules and guarded commands. 4. A module that implements the algorithm to translate an arena to its induced Kripke structure, represented as a Mr.Waffles `PredicatedGraph` instance. 5. A module that implements a construction to translate an arena, given as a list of modules, into a single CTL formula (used with the Ctl Sat command-line interface) representing the branching behaviour of the arena; this module is also responsible for wrapping the Ctl Sat command-line interface, using a Python subprocess instance.

*Evaluation.* EAGLE was tested with a number of systems taken from the literature, and the results are reported in [18]. The running time measures show that its performance is greatly driven by the CTL satisfiability solver, which is used to check whether an alternative player's strategy could be constructed whenever a strategy profile does not satisfy some player's goal. Details can be found in [18]. These experimental results go from two-player games that required hours to be analysed (Ctl Sat used) to multi-player games whose equilibrium analysis took a few seconds (only Ctl MC used). It was clear, in all cases, that the bottleneck was in the CTL satisfiability subroutine. In the future, we would like to compare EAGLE with PRALINE [4], the only other tool we are aware of that is focused on the equilibrium analysis of concurrent games.

*Example.* This example illustrates the concrete syntax used for modules in SRML as well as its translation to the concrete syntax in our Python implementation. The SRML module depicted below (on the left), 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 $(\top, \bot)$ or the value $(\bot, \top)$. The first **update** command says that if $(x, y)$ has the value $(\top, \bot)$ then the corresponding choice is to assign it the value $(\bot, \top)$, while the second command says that if the pair $(x, y)$ has the value $(\bot, \top)$, 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 *non-deterministically* picks an initial pair in $\{(\top, \bot), (\bot, \top)\}$, then at each round it *deterministically* toggles between these two pairs. This SRML module is written in our Python assistant for equilibrium checking as shown below (on the right):

**module** *toggle* **controls** $x, y$
  **init**
  $:: \top \rightsquigarrow x' := \top,\ y' := \bot$
  $:: \top \rightsquigarrow x' := \bot,\ y' := \top$
  **update**
  $:: (x \wedge \neg y) \rightsquigarrow x' := \bot,\ y' := \top$
  $:: (\neg x \wedge y) \rightsquigarrow x' := y,\ y' := x$

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

## 5 Case Study: A Peer-to-Peer Communication Protocol

To understand better the usefulness of an equilibrium checking tool, we now present a case study based on the system presented in [8]. Consider a peer-to-peer network with two agents (the extension to $n > 2$ agents is straightforward—we restrict to two agents only due to space and ease of presentation). 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 [8] considers an iBG model [10], where there are no constraints on the values that players choose for the variables under their control, we will consider a modified version of the communication protocol: using guarded commands, we require that an agent cannot both download and upload at the same time. This is a simple example of a system which cannot be specified as an iBG, but which has an SRML representation.

We can specify the game modelling the above communication protocol as a game 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 a game $G = (A, \gamma_0, \gamma_1)$, where $A = (\{0, 1\}, \Phi, m_0, m_1)$, $\Phi = \{u_0, u_1, d_0, d_1\}$, and $m_0, m_1$ are defined as follows:

**module** $m_0$ **controls** $u_0, d_0$
**init**
$:: \top \rightsquigarrow u_0' := \top,\ d_0' := \bot$
$:: \top \rightsquigarrow u_0' := \bot,\ d_0' := \top$
**update**
$:: \top \rightsquigarrow u_0' := \top,\ d_0' := \bot$
$:: \top \rightsquigarrow u_0' := \bot,\ d_0' := \top$

**module** $m_1$ **controls** $u_1, d_1$
**init**
$:: \top \rightsquigarrow u_1' := \top,\ d_1' := \bot$
$:: \top \rightsquigarrow u_1' := \bot,\ d_1' := \top$
**update**
$:: \top \rightsquigarrow u_1' := \top,\ d_1' := \bot$
$:: \top \rightsquigarrow u_1' := \bot,\ d_1' := \top$

Players' goals can be easily specified in CTL: the informal *"infinitely often"* requirement can be expressed in CTL as *"From all system states, on all paths, eventually"*. Hence, for $i \in \{0, 1\}$, we define the goals as follows: $\gamma_i = \mathbf{AGAF}(d_i \wedge u_{1-i})$.

This is clearly a very simple system/game: only two players and four controlled variables. Yet, checking the Nash equilibria of the game associated with this system is a hard problem. One can show—and formally verify using EAGLE—that this game has at least two different kinds of Nash equilibria (one where no player gets its goal achieved, and another one, which is Pareto optimal, where both players get their goal achieved). In general, the game has infinitely many Nash equilibria, but they all fall within the above two categories. Based on the SRML specifications of players' strategies given below, which can be seen to be consistent with modules $m_0$ and $m_1$, we can verify that both $(\mathit{StPlayer}(0), \mathit{StPlayer}(1)) \notin \mathit{NE}(G)$ and $(\mathit{OnlyUp}(0), \mathit{OnlyUp}(1)) \in \mathit{NE}(G)$.

**module** *StPlayer*$(i)$ **controls** $u_i, d_i$
**init**
$:: \top \rightsquigarrow u_i' := \top,\ d_i' := \bot$
**update**
$:: \top \rightsquigarrow u_i' := d_i,\ d_i' := u_i$

**module** *OnlyUp*$(i)$ **controls** $u_i, d_i$
**init**
$:: \top \rightsquigarrow u_i' := \bot,\ d_i' := \top$
**update**
$:: \top \rightsquigarrow u_i' := \bot,\ d_i' := \top$

## 6 Future Work

We see a number of ways in which EAGLE can be improved: From a theoretical point of view, there is no reason to restrict to CTL goals. More powerful temporal logics could be considered. Also, our tool solves games with respect to the most widely used solution concept in game theory: Nash equilibrium. However, other solution concepts could be considered. It would also be useful to support, *e.g.*, quantitative/probabilistic reasoning or epistemic specifications so that more general agent's preference relations or beliefs can be modelled. Finally, even though our verification system is quite easy to use, we could implement a more user-friendly interface to input temporal logic goals. At present, our main limitations are given by the syntax used by the two external libraries we use to solve the underlying CTL satisfiability and model checking problems.

**Acknowledgment.** EAGLE was implemented by Toumi as part of his final Computer Science project [18] at Oxford. Both EAGLE and [18] can be obtained from him. (To obtain EAGLE or [18], please, send an email to Alexis.Toumi at gmail.com). We also acknowledge the support of the ERC Research Grant 291528 ("RACE") at Oxford.

## References

1. Alur, R., Henzinger, T.A., Mang, F., Qadeer, S., Rajamani, S., Tasiran, S.: MOCHA: modularity in model checking. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
2. Alur, R., Henzinger, T.A.: Reactive modules. Form. Meth. Syst. Des. **15**(1), 7–48 (1999)
3. Berwanger, D., Chatterjee, K., De Wulf, M., Doyen, L., Henzinger, T.A.: Alpaga: a tool for solving parity games with imperfect information. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 58–61. Springer, Heidelberg (2009)
4. Brenguier, R.: PRALINE: a tool for computing nash equilibria in concurrent games. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 890–895. Springer, Heidelberg (2013)
5. Čermák, P., Lomuscio, A., Mogavero, F., Murano, A.: MCMAS-SLK: a model checker for the verification of strategy logic specifications. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 525–532. Springer, Heidelberg (2014)
6. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal Stratego. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 206–211. Springer, Heidelberg (2015)

7. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics, pp. 996–1072. Elsevier, Amsterdam (1990)
8. Fisman, D., Kupferman, O., Lustig, Y.: Rational synthesis. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 190–204. Springer, Heidelberg (2010)
9. Friedmann, O., Lange, M.: Solving parity games in practice. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 182–196. Springer, Heidelberg (2009)
10. Gutierrez, J., Harrenstein, P., Wooldridge, M.: Iterated boolean games. In: IJCAI, IJCAI/AAAI (2013)
11. Gutierrez, J., Harrenstein, P., Wooldridge, M.: Verification of temporal equilibrium properties of games on Reactive Modules. Technical report, University of Oxford (2015)
12. Kupferman, O., Vardi, M., Wolper, P.: Module checking. Inf. Comput. **164**(2), 322–344 (2001)
13. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
14. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: a model checker for the verification of multi-agent systems. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 682–688. Springer, Heidelberg (2009)
15. Osborne, M.J., Rubinstein, A.: A Course in Game Theory. MIT Press, Cambridge (1994)
16. Prezza, N.: CTLSAT (2015). https://github.com/nicolaprezza/CTLSAT
17. Reynaud, D., Mr. Waffles: (2015). http://mrwaffles.gforge.inria.fr
18. Toumi, A.: Equilibrium checking in Reactive Modules games. Technical report, Department of Computer Science, University of Oxford (2015)
19. van der Hoek, W., Lomuscio, A., Wooldridge, M.: On the complexity of practical ATL model checking. In: AAMAS, pp. 201–208. ACM (2006)