Automatic Verification of FSA Strategies via Counterexample-Guided Local Search for Invariants

Kailun Luo and **Yongmei Liu***
Dept. of Computer Science, Sun Yat-sen University, Guangzhou 510006, China
luokl3@mail2.sysu.edu.cn, ymliu@mail.sysu.edu.cn

Abstract

Strategy representation and reasoning has received much attention over the past years. In this paper, we consider the representation of general strategies that solve a class of (possibly infinitely many) games with similar structures, and their automatic verification, which is an undecidable problem. We propose to represent a general strategy by an FSA (Finite State Automaton) with edges labelled by restricted Golog programs. We formalize the semantics of FSA strategies in the situation calculus. Then we propose an incomplete method for verifying whether an FSA strategy is a winning strategy by counterexample-guided local search for appropriate invariants. We implemented our method and did experiments on combinatorial game and also single-agent domains. Experimental results showed that our system can successfully verify most of them within a reasonable amount of time.

1 Introduction

Strategy representation and reasoning has received much attention over the past years. The two most popular strategic logics are Alternating-time Temporal Logic (ATL) [Alur *et al.*, 2002] and Strategy Logic (SL) [Chatterjee *et al.*, 2010]. In ATL, strategies are treated implicitly: the formula $\langle\langle A\rangle\rangle\phi$ expresses that coalition A has a strategy to ensure ϕ hold. In SL, a strategy is an explicit first-order object which is a function from sequences of states to actions. Model-checking for ATL is in PTIME, while model-checking for SL is non-elementarily decidable. MCMAS [Lomuscio *et al.*, 2009] and MCMAS-SLK [Cermák *et al.*, 2014] are representative model checking tools for ATL and SL respectively.

In this paper, we are concerned about the representation and reasoning of general strategies that solve a class of (possibly infinitely many) games with similar structures, represented by a basic action theory in the situation calculus [Reiter, 2001]. A related problem is generalized planning where a single plan works for possibly infinitely many similar planning problems [Levesque, 2005; Srivastava *et al.*, 2008]. To illustrate our problem, consider the Chomp game. As shown in Figure 1, cookies are laid out on a rectangular grid of an arbitrary size.

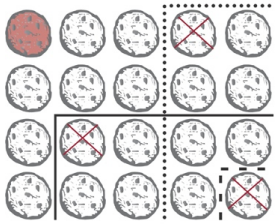

Figure 1: The Chomp game

The cookie in the top left position (1,1) is poisoned. Two players take turns making moves: at each move, a player eats a remaining cookie, together with all cookies to the right or below it. When the grid is square, a (general) winning strategy for player 1 is this: First, eat the cookie in position (2,2); then when the opponent eats the cookie in position (x, y), respond with eating the cookie in position (y, x).

To represent generalized plans that solve multiple planning instances, Hu and Levesque [2011] proposed FSA (Finite State Automaton) plans. Inspired by this work, in this paper, we propose to represent a general strategy by an FSA with edges labeled by restricted programs of Golog [Levesque *et al.*, 1997]. FSA representation of strategies has the advantages of ease to represent nested loop structures and amenability to algorithmic operations. We formalize the semantics of FSA strategies in the situation calculus, by making use of second-order logic.

In this paper, we explore the automatic verification of FSA strategies, in particular, verifying if an FSA strategy is a winning strategy wrt a basic action theory in the situation calculus. Since this is a (second-order) theorem-proving task, we propose a sound but incomplete method via counterexample-guided local search for appropriate invariants, which can be checked with first-order theorem proving. For an FSA strategy, an invariant is a labelling of each strategy state with a formula so that if the formula is true in the state, in any successor state, the label formula will be true. The method of counterexample-guided refinement is widely used in the formal methods community, *e.g.*, CEGAR (Counter-Example Guided Abstraction Refinement) [Clarke *et al.*, 2000; 2003]. The idea is: first generate an initial abstraction; if it does not

*Corresponding author

serve the purpose, we get a counterexample and use it to refine our abstraction, and then repeat the process.

We implemented our method and experimented on two-player domains from the combinatorial game literature. We also tested our method with single-agent domains from the planning literature by adapting them to our framework. Our system can successfully verify most of them within a reasonable amount of time.

2 Preliminaries

In this section, we introduce the situation calculus, Golog programs, regression, and small model progression.

The situation calculus is a many-sorted first-order logical language with limited second-order features [Reiter, 2001]. In a situation calculus language, there are three disjoint sorts: $action$ for actions, $situation$ for situations and $object$ for everything else. The symbol S_0 denotes the initial situation; $do(a, s)$ is a binary function representing the situation resulting from performing action a in situation s; the relation $Poss(a, s)$ states that it is possible to perform action a in situation s. There are action functions, *e.g.*, $put(x, y)$. A fluent is a special relation/function whose values vary from situation to situation and is denoted by a relation/function whose last argument is a situation term. If a formula refers to a particular situation τ, we call it uniform in τ. A uniform formula ϕ with all situation arguments removed is called a *situation-suppressed* formula, and $\phi[s]$ denotes the formula obtained from ϕ by restoring s as the situation arguments to all fluents.

In the situation calculus, a particular domain of application is specified by a basic action theory (BAT) of the form:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}, \text{where}$$

1. Σ is the set of the foundational axioms for situations.
2. $\mathcal{D}_{ap}$ contains a single precondition axiom of the form $Poss(a, s) \equiv \Pi(a, s)$, where $\Pi(a, s)$ is uniform in s.
3. $\mathcal{D}_{ss}$ is a set of successor state axioms (SSAs), one for each relational fluent, of the form
$$F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s),$$
and one for each functional fluent, of the form
$$f(\vec{x}, do(a, s)) = y \equiv \Phi_f(\vec{x}, y, a, s),$$
where $\Phi_F(\vec{x}, a, s)$ and $\Phi_f(\vec{x}, y, a, s)$ are uniform in s.
4. $\mathcal{D}_{una}$ is the set of unique names axioms for actions.
5. $\mathcal{D}_{S_0}$, the initial KB, is a set of sentences uniform in S_0.

Golog is a high-level programming language for representing complex actions. The formal semantics of Golog is specified by an abbreviation $Do(\delta, s, s')$, meaning that situation s will be updated to s' after executing program δ. $Do(\delta, s, s')$ is inductively defined as follows:

1. Primitive actions: For any action term α,
$Do(\alpha, s, s') \doteq Poss(\alpha, s) \wedge s' = do(\alpha, s)$.
2. Test actions: For any situation-suppressed formula ϕ,
$Do(\phi?, s, s') \doteq \phi[s] \wedge s = s'$.
3. Sequence:
$Do(\delta_1; \delta_2, s, s') \doteq \exists s''.Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$.
4. Nondeterministic choice of two actions:
$Do(\delta_1|\delta_2, s, s') \doteq Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$.
5. Nondeterministic choice of action arguments:
$Do((\pi x)\delta(x), s, s') \doteq (\exists x)Do(\delta(x), s, s')$.
6. Nondeterministic iteration:
$Do(\delta^*, s, s') \doteq (\forall P).\{(\forall s_1)P(s_1, s_1) \wedge (\forall s_1, s_2, s_3)$
$[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)]\} \supset P(s, s')$.

Regression is an important computational mechanism for reasoning about actions. Here we present the one step regression operator.

Definition 1 Given a BAT $\mathcal{D}$, we use $\mathcal{R}[\phi]$ to denote the formula obtained from ϕ by the following steps:

1. For each functional fluent term $f(\vec{t}, do(\alpha, \sigma))$, replace the current formula ψ with $(\exists y).\Phi_f(\vec{t}, y, \alpha, \sigma) \wedge \psi[f(\vec{t}, do(\alpha, \sigma))/y]$, where $\psi[t/y]$ denotes the result of replacing all occurrences of t in ψ by y.
2. Replace each fluent atom $F(\vec{t}, do(\alpha, \sigma))$ with $\Phi_F(\vec{t}, \alpha, \sigma)$.
3. Replace each atom $Poss(\alpha, \sigma)$ with $\Pi(\alpha, \sigma)$.
4. Further simplify the result by using $\mathcal{D}_{una}$.

Proposition 1 $\mathcal{D} \models \phi \equiv \mathcal{R}[\phi]$.

Li and Liu [2015] extended the regression of formulas wrt primitive actions to that wrt programs. Here we omit the case of iteration since we do not need it in this paper.

Definition 2 Let $\phi(s)$ be a formula uniform in s. The regression of $\phi(s)$ wrt program δ, written $\mathcal{R}[\phi(s), \delta]$, is defined as:

- $\mathcal{R}[\phi(s), \alpha] \doteq \mathcal{R}[Poss(\alpha, s) \supset \phi(do(\alpha, s))]$.
- $\mathcal{R}[\phi(s), \psi?] \doteq \psi[s] \supset \phi(s)$.
- $\mathcal{R}[\phi(s), \delta_1; \delta_2] \doteq \mathcal{R}[\mathcal{R}[\phi(s), \delta_2], \delta_1]$.
- $\mathcal{R}[\phi(s), \delta_1|\delta_2] \doteq \mathcal{R}[\phi(s), \delta_1] \wedge \mathcal{R}[\phi(s), \delta_2]$.
- $\mathcal{R}[\phi(s), (\pi x)\delta(x)] \doteq (\forall x)\mathcal{R}[\phi(s), \delta(x)]$.

Based on Proposition 1, by induction, it is easy to prove:

Proposition 2 *Let δ be a Golog program not involving nondeterministic iteration. We have:*

$$\mathcal{D} \models \forall s.\mathcal{R}[\phi(s), \delta] \equiv \forall s'.Do(\delta, s, s') \supset \phi(s').$$

A small model can be represented as a finite set of ground atoms under the closed-world assumption. Li and Liu [2015] presented the notion of small model progression, and use $prog[M, \delta]$ to denote the set of small models resulting from M by executing δ. The formal definition is as follows:

Definition 3 Given a small model M with domain D and a program δ, the progression of M wrt δ, denoted as $prog[M, \delta]$, results in a set of small models:

- $prog[M, \alpha] =$ 1. $\emptyset$ if $M[s] \not\models Poss(\alpha, s)$.
2. $\{M'\}$ if $M[s] \models Poss(\alpha, s)$, where M' is the model obtained from M by applying the effects of α.
- $prog[M, \psi?] =$ 1. $\emptyset$ if $M[s] \not\models \psi[s]$.
2. $\{M\}$ if $M[s] \models \psi[s]$.
- $prog[M, \delta_1; \delta_2] = prog[prog[M, \delta_1], \delta_2]$.
- $prog[M, \delta_1|\delta_2] = prog[M, \delta_1] \cup prog[M, \delta_2]$.
- $prog[M, (\pi x)\delta(x)] = \bigcup\{prog[M, \delta(c)]|c \in D\}$.

When $\mathcal{M}$ is a set of small models, we define
$prog[\mathcal{M}, \delta] = \bigcup\{prog[M, \delta]|M \in \mathcal{M}\}$.

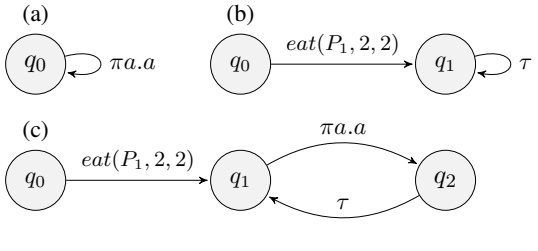

Figure 2: An FSA strategy and its composite strategy

3 FSA Strategies

In this section, we formalize the concept of FSA strategies in the situation calculus. We focus on combinatorial games [Ferguson, 2014], *i.e.*, two-player turn-based games which always end in a finite number of moves.

We make the following extensions of the situation calculus and a basic action theory $\mathcal{D}$: We introduce a new sort *state*, a countable infinite set of state constants $\mathcal{Q} = \{Q_0, Q_1, \ldots\}$, and unique name axioms of the form $Q_i \neq Q_j$, for $i \neq j$. We introduce a new sort $player$, two player constants P_1 and P_2, together with the axiom $\forall p.(p = P_1 \vee p = P_2) \wedge (P_1 \neq P_2)$. We use a relational fluent $turn(p, s)$, meaning in situation s, it is p's turn to play. We add the axioms: $turn(p, do(a, s)) \equiv \neg turn(p, s)$ and $turn(P_1, S_0) \wedge \neg turn(P_2, S_0)$. We introduce abbreviations $end(s)$, meaning the game ends in situation s, and $win(p, s)$, meaning p wins in situation s.

We add a special action $\pi a.a$, meaning doing any executable action. Its formal semantics is defined as follows:

$Do(\pi a.a, s, s') \doteq \exists a.Poss(a, s) \wedge s' = do(a, s)$.

We now formalize the chomp game.

Example 1 We use two constants N and M to denote the size of the rectangular grid. Fluent $ch(x, y, s)$ means there is a cookie at position (x, y). Action $eat(p, x, y)$ means player p eats the cookie at position (x, y) together with all cookies to the right or below it. Fluent $last(x, y, s)$ means the last action is $eat(p, x, y)$ for some player p. In addition to the second-order axiomatization of Peano arithmetic, we have the following axioms:

$$Poss(eat(p, x, y), s) \equiv turn(p, s) \wedge ch(x, y, s)$$
$$ch(x, y, do(a, s)) \equiv$$
$$\exists p, i, j.a = eat(p, i, j) \wedge (i > x \vee j > y)$$
$$last(x, y, do(a, s)) \equiv \exists p.a = eat(p, x, y)$$
$$ch(x, y, S_0) \equiv 0 < x \leq N \wedge 0 < y \leq M$$
$$win(p, s) \doteq turn(p); \qquad end(s) \doteq \neg ch(1, 1, s)$$

We use Chomp NxN (resp. 2xN) to represent the chomp game with square (resp. two row) grids, and add the Initial Situation Axiom $M = N$ (resp. $M = 2$).

Note that although each chomp game is finite, since there is no bound on the size of the game, there are infinitely many chomp games.

We say that a Golog program is $single\text{-}step$ if its execution always results in the execution of a single primitive action.

Definition 4 An FSA strategy is a tuple $S = (\mathbf{Q}, \mathbf{E}, q_0)$ s.t. $\mathbf{Q}$ is a finite set of states, and $q_0 \in \mathbf{Q}$ is the initial state; $\mathbf{E}$ is a finite set of edges of the form $q \xrightarrow{\tau} q'$, where q is the starting state, q' is the ending state, and τ is the label, which is a single-step Golog program.

A notable difference from FSA plans is that we use single-step Golog programs as edge labels. This allows for more compact representation of strategies.

We say that S is a strategy for player p if all primitive actions are actions of p. We say that S is a $compositestrategy$ if for any state q, all primitive actions ending at q are actions of a player and all primitive actions leaving q are actions of the other player.

Intuitively, a state of an FSA strategy S for a player p represents a mental state of p for decision making. The execution of S is as follows. Initially, p is in state q_0. When the player is in state q, she can choose an edge $q \xrightarrow{\tau} q'$, perform τ and move to q'. Intuitively, a state of an FSA composite strategy C represents mental states of both players.

Figure 2(a) shows an FSA strategy with a single state q_0 and a single edge from q_0 to q_0 labeled with $\pi a.a$. It indicates the strategy for a player where the player simply takes any executable action. We call such a strategy the $null\ strategy$. Strategies for both players can be composed to form a composite strategy, as defined below:

Definition 5 Given an FSA strategy $S_1 = (\mathbf{Q}_1, \mathbf{E}_1, q_0^1)$ for player 1 and an FSA strategy $S_2 = (\mathbf{Q}_2, \mathbf{E}_2, q_0^2)$ for player 2, we define their composition strategy $S = (\mathbf{Q}, \mathbf{E}, q_0)$ as follows: $\mathbf{Q} = \mathbf{Q}_1 \times \mathbf{Q}_2 \times \{1, 2\}$, $q_0 = (q_0^1, q_0^2, 1)$, and for any $q_1 \in \mathbf{Q}_1$, $q_2 \in \mathbf{Q}_2$, $(q_1, q_2, 1) \xrightarrow{\tau} (q_1', q_2, 2)$ iff $q_1 \xrightarrow{\tau} q_1'$ in S_1, and $(q_1, q_2, 2) \xrightarrow{\tau} (q_1, q_2', 1)$ iff $q_2 \xrightarrow{\tau} q_2'$ in S_2.

Example 1 cont'd Figure 1(b) shows an FSA strategy of player 1 for the Chomp NxN game. Player 1 begins with action $eat(P_1, 2, 2)$ and moves to state q_1. In state q_1, for P_2's action $eat(P_2, x, y)$, player 1 responds with $eat(P_1, y, x)$ and stays in state q_1. Figure 1(c) is the result of composing the above strategy with the null strategy.

In the following, we formalize the semantics of FSA strategies. Given an FSA strategy S, we first introduce a transition relation $T_S(q, s, q', s')$, meaning that from situation s, it is possible to reach situation s' by executing a program starting at state q and ending at state q'.

Definition 6 Given an FSA strategy $S = (\mathbf{Q}, \mathbf{E}, Q_0)$ where $\mathbf{Q} \subseteq \mathcal{Q}$, we use $T_S(q, s, q', s')$ as abbreviation for the following formula: $\bigvee_{g \xrightarrow{\tau} g' \in \mathbf{E}} g = q \wedge g' = q' \wedge Do(\tau, s, s')$.

For a composite strategy S, we define the reflexive transitive closure of T_S : intuitively, $T_S^*(q, s, q', s')$ means that from situation s, it is possible to reach situation s' by following the composite strategy.

Definition 7 Given an FSA composite strategy $S = (\mathbf{Q}, \mathbf{E}, Q_0)$ where $\mathbf{Q} \subseteq \mathcal{Q}$, we use $T_S^*(q, s, q', s')$ as abbreviation for $\forall R.\{\phi \supset R(q, s, q', s')\}$, where ϕ is the conjunction of the universal closure of

- $R(q, s, q, s)$, and
- $R(q, s, q', s') \wedge T_S(q', s', q'', s'') \supset R(q, s, q'', s'')$.

It is possible that a strategy is incomplete in the sense that when the opponent adopts the null strategy, there might be

situations where the player is in a state q where no program starting at q is executable.

Definition 8 Given a BAT $\mathcal{D}$ and an FSA strategy S for player p, we say S is complete if the composition C of S and the null strategy satisfies $\mathcal{D} \models$

$$\forall q, s. T_C^*(Q_0, S_0, q, s) \wedge \neg end(s) \supset \exists q', s'. T_C(q, s, q', s').$$

It is easy to make an incomplete strategy complete as follows. We add a special dead state. For each state q, we add an edge from it to the dead state labeled with the program $\phi_q?; \pi a.a$, where ϕ_q is the negation of the disjunction of the executability condition of each program leaving q.

We now define the concept of winning strategies. Intuitively, a strategy for player p is a winning strategy if Cond1: p always wins no matter what strategy the opponent adopts, which is equivalent to Cond2: p always wins when the opponent adopts the null strategy. To see why, Cond1 obviously implies Cond2, and Cond2 implies Cond1 since for any strategy the opponent adopts, the set of reachable game states is a subset of that when the opponent adopts the null strategy.

Definition 9 Given a BAT $\mathcal{D}$ and a complete FSA strategy S for player p, we say S is a winning strategy if the composition C of S and the null strategy satisfies

$$\mathcal{D} \models \forall q, s. T_C^*(Q_0, S_0, q, s) \wedge end(s) \supset win(p, s).$$

Note that we only consider games which always ends in a finite number of moves. Thus in the definition, we do not need to require that the strategy must ensure the game ends.

We stress that to verify if an FSA strategy is a winning strategy wrt a BAT $\mathcal{D}$, representing a class of (possibly infinitely many) games, we have a theorem-proving task. Since $\mathcal{D}$ includes a second-order induction axiom for situations and T_C^* is a second-order formula, the verification of winning strategies is a second-order reasoning task.

In fact, the winning property of strategies is an example of safety properties, asserting that a formula ϕ holds in any reachable game state if the strategy is adopted, which can be formalized as follows: $\mathcal{D} \models \forall q, s. T_C^*(Q_0, S_0, q, s) \supset \phi(s)$.

4 Theoretic Foundations

In this section, we introduce the theoretic foundations for our method for automatic verification of FSA strategies.

We first define the concept of the winning condition of state q for player p. Intuitively, it means the condition that starting from q, whenever the game ends, p wins.

Definition 10 Let C be a composite strategy. The winning condition of state q for player p, written $W(p, q, s)$, is defined as the formula:

$$\forall q', s'. T_C^*(q, s, q', s') \wedge end(s') \supset win(p, s').$$

Thus a complete strategy S is a winning strategy for player p if $\mathcal{D} \models W(p, Q_0, S_0)$.

The concept of loop invariants plays an important role in program verification. It can be naturally extended to FSA strategies and employed to prove that $\mathcal{D} \models W(p, Q_0, S_0)$.

We will make use of two kinds of labeling of strategy states. The first kind labels each state with a formula, and we call it an $F\text{-}labeling$. The second kind labels each state with a set of small models, and we call it an $M\text{-}labeling$.

Definition 11 Given a BAT $\mathcal{D}$ and an FSA composite strategy C, we say an F-labeling X of C is an $invariant$ of C, if for any edge $q \xrightarrow{\tau} q'$, we have

$$\mathcal{D} \models \forall s, s'. X(q)[s] \wedge Do(\tau, s, s') \supset X(q')[s'].$$

So an invariant of C is a labeling of each state of C with a formula such that if the formula is true in the state, in any successor state, its label formula will also be true.

Definition 12 Given a BAT $\mathcal{D}$, an FSA composite strategy C, and an invariant X of C,

1. we say X is sufficient if for any state q, we have $\mathcal{D} \models \forall s. X(q)[s] \wedge end(s) \supset win(p, s)$;
2. we say X is necessary if $\mathcal{D}_{S_0} \models X(Q_0)[S_0]$.

Proposition 3 Let X be a sufficient invariant of composite strategy C. Then for any state q, we have $\mathcal{D} \models \forall s. X(q)[s] \supset W(p, q, s)$. If X is also necessary, then the strategy for p is a winning strategy.

Proof: We prove that for any state q, we have $\mathcal{D} \models \forall s, q', s'. X(q)[s] \wedge T_C^*(q, s, q', s') \wedge end(s') \supset win(p, s')$. Let M be a model of $\mathcal{D}$. We prove that $M \models X(q)[s] \wedge T_C^*(q, s, q', s') \wedge end(s') \supset win(p, s')$ for any q, s, q', s'. We prove by induction on the distance d between s and s'. Basis: $d = 0$ hence $q' = q$ and $s' = s$. By the given condition, we have $M \models X(q)[s] \wedge end(s) \supset win(p, s)$. Thus $M \models win(p, s)$. Induction: We assume the proposition holds when $d = n$, and proceed to prove that it holds when $d = n + 1$. So suppose $M \models X(q)[s] \wedge T_C^*(q, s, q', s') \wedge end(s')$ and $dist(s, s') = n + 1$. Then there exist q'' and s'' such that $M \models T_C(q, s, q'', s'') \wedge T_C^*(q'', s'', q', s')$ and $dist(s'', s') = n$. By the definition of T_C, there is an edge $q \xrightarrow{\tau} q''$ s.t. $M \models Do(\tau, s, s'')$. Since X is an invariant, $M \models X(q'')(s'')$. Now by induction, $M \models win(p, s')$.

Thus $\mathcal{D} \models X(Q_0)[S_0] \supset W(p, Q_0, S_0)$. If X is also necessary, $\mathcal{D}_{S_0} \models X(Q_0)[S_0]$. It follows $\mathcal{D} \models W(p, Q_0, S_0)$. ■

In the following, we first show how to verify if an F-labeling is an invariant by using the idea of regression. Then we will analyze properties of necessary invariants.

Definition 13 Let C be an FSA composite strategy and X an F-labeling of C. The regression of X wrt C, written $\mathcal{R}(X, C)$, is a new F-labeling X' defined as follows: for each state q, $X'(q) = \bigwedge_{q \xrightarrow{\tau} q'} \mathcal{R}(X(q'), \tau)$.

By Proposition 2 from Preliminaries, we have

Proposition 4 *Let X be an F-labeling of C. If $\mathcal{D} \models X \supset \mathcal{R}(X, C)$, meaning $\mathcal{D} \models \forall s. X(q)[s] \supset \mathcal{R}(X, C)(q)[s]$ for each state q, then X is an invariant of C.*

We now extend the notion of small model progression wrt a program to that wrt a strategy.

Definition 14 Let C be an FSA composite strategy and L an M-labeling of C. The progression of L wrt C, written $\mathcal{P}(L, C)$, is a new M-labeling defined as follows: for each state q', $L'(q') = L(q') \cup \bigcup_{q \xrightarrow{\tau} q'} prog(L(q), \tau)$.

Definition 15 Let C be an FSA composite strategy and M a small model. We inductively define a sequence L_n, $n \geq 0$, of M-labelings as follows:

- $L_0(q_0) = \{M\}$, and $L_0(q) = \emptyset$ for all state $q \neq q_0$;
- $L_{n+1} = \mathcal{P}(L_n, C)$, for $n \geq 0$.

We call L_n the nth progression of M wrt C, and denote it by $\mathcal{P}^n(M, C)$.

Intuitively, $M' \in \mathcal{P}^n(M, C)(q)$ iff starting from M at state q_0, by following the strategy for at most n steps, M' is a possible model for state q.

Proposition 5 *Given a BAT $\mathcal{D}$, let X be a necessary invariant of a composite strategy C. Let $M_0[S_0] \models \mathcal{D}_{S_0}$, and $L_n = \mathcal{P}^n(M_0, C)$, $n \geq 0$. Then for any state q, for any $M \in L_n(q)$, $M \models X(q)$.*

Proof: We prove by induction on n. Basis: $n = 0$. Since $M_0[S_0] \models \mathcal{D}_{S_0}$ and $\mathcal{D}_{S_0} \models X(Q_0)[S_0]$, $M_0 \models X(Q_0)$. Induction: $L_{n+1} = \mathcal{P}(L_n, C)$. It suffices to prove that for each $q \xrightarrow{\tau} q'$, if $M \models X(q)$, then $M' \models X(q')$, where $M' = prog(M, \tau)$. Since X is an invariant, $\mathcal{D} \models \forall s, s'.X(q)[s] \wedge Do(\tau, s, s') \supset X(q')[s']$. Since $M[s], M[s'] \models Do(\tau, s, s')$, we get if $M \models X(q)$, then $M' \models X(q')$. ■

In this paper, we consider invariants of the form $(end(s) \supset win(p, s)) \wedge \phi$, where ϕ is a CNF formula whose free variables are implicitly universally quantified. We use counterexample-guided local search for sufficient and necessary invariants. Below, we give the relevant definitions.

A literal is an atom or the negation of an atom. A clause is a set of literals, understood as their disjunction. If a clause has free variables, they are implicitly universally quantified. For example, the clause $c : \neg P(x, y) \vee \neg P(y, z) \vee P(x, z)$ is understood as $\forall x, y, z.c$. The length of a clause c, written $|c|$, is the number of literals in c. A CNF is the conjunction of clauses. A CNF is considered to have a default clause $\top$.

Definition 16 Let c be a clause, and $\mathcal{M}^+$ and $\mathcal{M}^-$ sets of models. We say that c characterizes $\mathcal{M}^+$ vs $\mathcal{M}^-$ if each model of $\mathcal{M}^+$ satisfies c and each model of $\mathcal{M}^-$ falsifies it.

Definition 17 Let c_1 and c_2 be two clauses. The distance between c_1 and c_2 is defined as follows: $dist(c_1, c_2) = \max\{|c_1 - c_2|, |c_2 - c_1|\}$, where "$-$" is set difference. Let k be a positive integer. A k-neighbor of a clause c is a clause c' s.t. their distance is bounded by k.

5 Automatic Verification of FSA Strategies

In this section, we present our method for automatic verification of FSA strategies.

The main idea of our method is as follows. Let S be a complete FSA strategy and C its composition with the null strategy. To prove that S is a winning strategy, we attempt to find a sufficient and necessary invariant X of C. As the initial value of X, we label each state with $end(s) \supset win(p, s)$. We check if X is an invariant, *i.e.*, if $\models X \supset \mathcal{R}(X, C)$. If not, we obtain counterexamples and use them to do a local update of X to exclude the counterexamples. The update may fail. In case of failure, we restart with $end(s) \supset win(p, s)$. If X is an invariant, we check if it is necessary, *i.e.*, if $\mathcal{D}_{S_0} \models X(Q_0)[S_0]$. If so, we return yes. Otherwise, we obtain a counterexample and we compute the nth progression of it wrt C. Now for each state q, we do a local update of $X(q)$ to include the small models associated to q. Then we repeat the process.

Algorithm 1: $verify(\mathcal{D}, S, p)$

```
C := the composition of S with the null strategy
Θ := genPreds(D, S)
L⁻, L⁺ := Label[C, ∅]
X := Label[C, end(s) ⊃ win(p, s)]
while not timing-out do
    while ⊭ X ⊃ R(X, C) do
        foreach state q s.t. there is a counterexample
          M_q of X(q) ⊃ R(X, C)(q) do
            L(q) := {M_q}
        update⁻(X, L, L⁻, L⁺)
        if update⁻ fails then reset(X, L⁻, L⁺);
    if D_{S0} ⊨ X(Q0)[S0] then return yes ;
    else
        M := a counterexample of D_{S0} ⊃ X(Q0)[S0]
        L := P^n(M, C) for a sufficiently large n
        update⁺(X, L, L⁻, L⁺)
return unknown
```

Algorithm 2: $update^-(X, L, L^-, L^+)$

```
foreach state q s.t. L(q) ≠ ∅ do
    let X(q) = (end(s) ⊃ win(p, s)) ∧ c1 ∧ ... ∧ cn
    c' := a maximal k-neighbor of some c_i s.t. c'
          characterizes L⁺(q) versus L⁻(q)(c_i) ∪ L(q)
    if such a c' does not exist then return failure ;
    replace c_i in X(q) with c'
    L⁻(c')(q) := L⁻(c_i)(q) ∪ L(q)
return success
```

Our verification method is presented by Algorithm 1. We first generate from $\mathcal{D}$ and S the set of predicates used to form the X labels of states. We make use of two M-labelings of C: for each state q, $L^-(q)$ is a set of models that $X(q)$ should exclude, and $L^+(q)$ is a set of models that $X(q)$ should include. The X label of each state is in the form of $(end(s) \supset win(p, s)) \wedge \phi$, where ϕ is a CNF. Thus for each state q, $L^-(q)$ is actually stored in the form of $L^-(q)(c)$ for each clause of $X(q)$, where $L^-(q)(c)$ is a set of models that falsify c. As the initial values of L^- and L^+, we label each state with the empty set. As the initial value of X, we label each state of C with $end(s) \supset win(p, s)$. The $update^-$ (resp. $update^+$) procedure is presented by Algorithm 2 (resp. 3).

To update the X label of a state, for a clause, we choose a maximal k-neighbour of it. We define a preference relation over clauses so that after reset, we will make choices different from those made earlier. The definition is through a scoring function. Initially, the score of each literal is 0. During reset, we decrease by one the score of each literal appearing in a clause which is replaced since the last reset. The score of a clause is the sum of the scores of the literals in the clause. A clause c_1 is preferred over another clause c_2 if $score(c_1) >$

Algorithm 3: $update^+(X, L, L^-, L^+)$

```
1 foreach state q s.t. L(q) ≠ ∅ do
2     let X(q) = (end(s) ⊃ win(p, s)) ∧ c_1 ∧ ... ∧ c_n
3     L^+(q) := L^+(q) ∪ L(q)
4     foreach c_i do
5         c'_i := a maximal k-neighbor of c_i s.t. c'_i
                  characterizes L^+(q) versus L^-(q)(c_i)
6         if such a c'_i does not exist then
7             c'_i := ⊤; L^-(q)(c_i) := ∅;
8         replace c_i in X(q) with c'_i
```

$score(c_2)$ or $score(c_1) = score(c_2)$ and $|c_1| < |c_2|$.

We emphasize that our algorithm is deterministic since we do not use randomization: the reason we find different invariants after restart is that when we restart, we update the score function and hence the preference relation between clauses.

Note that in Alg. 1 (Lines 6 and 11), we use the SMT solver Z3 [de Moura and Bjørner, 2008] for checking first-order entailments. If Z3 is unable to decide within the given time, it returns "unknown". In such a case, we will backtrack. If the last update of X occurs in a call to $update^-$ (resp. $update^+$), for each state q, we make another choice to update $X(q)$.

Finally, the $genPreds$ process is defined as follows. Extract from the given BAT $\mathcal{D}$ and strategy S the set $\mathcal{C}$ of constant symbols, set $\mathcal{F}$ of function symbols including both situation-independent functions and functional fluents, and set $\mathcal{P}$ of predicate symbols. Our basic idea of predicate generation is to generate predicates whose arguments are terms such as $P(x)$ and $P(f(x))$. However, to control the number of generated predicates, we only generate elementary predicates, that is, predicates of the form $t = f(\vec{t})$ or $P(\vec{t})$, where f is a function symbol, P is a predicate symbol, and t is a constant or variable. In fact, any predicate can be expressed as a conjunction of elementary predicates, for example, $P(f(x))$ can be expressed as $P(y) \wedge y = f(x)$. To further control the number of generated predicates, we use at most $m \geq 2$ different variables $x_1, \ldots, x_m$ in each generated predicate. We use symmetry and other properties to reduce the number of arithmetic predicates generated, *e.g.*, we generate the predicate $x = y + 1$ but not $x = y + 0$ or $x = 1 + y$.

Clauses are generated as follows: Each clause uses at most $n \geq 3$ variables $x_1, \ldots, x_n$; Atoms are obtained from generated predicates by applying injections from variables into variables. For example, from a generated predicate $P(x, y)$, we can get the clause $\neg P(x, y) \vee \neg P(y, z) \vee P(x, z)$.

Example 1 cont'd We extract the following symbols:
$\mathcal{C} = \{0, 1, 2, P_1, P_2, N, M\}$, and
$\mathcal{P} = \{turn(p), x > y, x \geq y, ch(x, y), last(x, y)\}$.

Thus examples of generated predicates are: $turn(P_1)$, $turn(p), x > 0, x > y, 2 > x, N > 2, ch(x, y), ch(1, 1)$.

Theorem 1 Given a BAT $\mathcal{D}$ and a complete strategy S for player p, when Alg. 1 returns yes, X is a sufficient and necessary invariant, hence S is a winning strategy.

In fact, Algorithm 1 can be easily generalized for verifying safety properties: to verify that a formula ϕ always holds, we replace $end(s) \supset win(p, s)$ in the algorithm with $\phi(s)$.

Example 1 cont'd We now illustrate the running process of our algorithm with the strategy shown in Figure 2. We only show the evolution of the label of state q_1:

After the first call to $update^-$, we get:
$X(q_1) : [\neg ch(1, 1) \supset turn(P_1)] \wedge [\neg N > x]$

After several calls to $update^-$ and $update^+$, we have:
$X(q_1) : [\neg ch(1, 1) \supset turn(P_1)] \wedge [\neg ch(x, x) \vee N \neq x]$
$\wedge [0 \geq x \vee 0 \geq y \vee \neg ch(x, y) \vee ch(y, x)]$

Now a counterexample M is generated:
$\{\langle ch(1, 1), ch(1, 2), ch(1, 3), ch(1, 4), ch(2, 1), ch(3, 1), ch(4, 1), turn(P_2), N = 4\rangle\}$

Calling $update^-$, we add a new clause $[N = x \vee x \neq 2]$.

Now we get a sufficient invariant, but when we check if $\mathcal{D}_{S_0} \models X(Q_0)[S_0]$, we get a counterexample M':
$\{\langle ch(1, 1), ch(1, 2), ch(1, 3), ch(2, 1), ch(2, 2), ch(2, 3), ch(3, 1), ch(3, 2), ch(3, 3), N = 3\rangle\}$

After doing model progression, we have $L(q_2)$:
$\{\langle ch(1, 1), ch(1, 2), ch(1, 3), ch(2, 1), ch(3, 1), N = 3\rangle, \langle ch(1, 1), ch(1, 2), ch(2, 1), N = 3\rangle, \langle ch(1, 1), N = 3\rangle\}$

Calling $update^+$, we replace $N = x \vee x \neq 2$ by $x \geq N \vee 2 \geq x$. ...

Finally, we get a sufficient and necessary invariant:
$X(q_1) : [\neg ch(1, 1) \supset turn(P_1)]$
$\wedge [0 \geq x \vee 0 \geq y \vee \neg ch(x, y) \vee ch(y, x)]$
$\wedge [\neg ch(x, x) \vee 2 > x] \wedge [\neg last(x, 0)]$

Thus the given strategy is a winning strategy.

6 Experimental Results

We implemented our algorithm using Python and Z3. All experiments were conducted on a Linux machine with 3.50GHz CPU and 16GB RAM. We set the implementation parameters as follows: The time-out bound for Z3 is 10 second; when updating the X-label, we use 2-neighbors of clauses; when generating predicates we use at most 2 variables and when generating clauses we use at most 4 variables. We did experiments on two types of domains: one from the combinatorial game literature and the other from the generalized planning literature, and we adapt the second type into our framework by adding an agent who is always doing the null action.

The following are two-agent domains and strategies to verify. The domains may look similar in that they all involve placing or removing tokens from finite grids. However, even board games can be very different: difference in game rules or initial settings lead to difference between games. For instance, the winning strategies for Chomp2xN and ChompNxN are quite different, thus the set of reachable sates and hence invariants are different.

PickStone123 (134): There are n stones and two players take turns to remove 1, 2 or 3 (resp. 1, 3 or 4) stones. The one who has no stones to remove loses the game. The strategy for player 1 is to remove stones so that the number of remaining stones satisfies $n\%4 = 0$ (resp. $n\%7 = 2 \vee n\%7 = 0$).

Chomp2xN (NxN): For Chomp2xN, the strategy for player 1 is this: first, eat the cookie in position $(2, N)$; when player 2 eats the cookie in position $(1, x)$ (resp. $(2, x)$), respond with eating the cookie in position $(2, x\text{-}1)$ (resp. $(1, x\text{+}1)$).

Clobber: On a 2xN grid, where N is even, white tokens are

Name	C	P	U^-	U^+	B	R	T(s)
PickS123	3	59	3	3	0	0	7.3
PickS134	3	65	3	3	0	0	10.6
chp 2×N	4	41	46	17	5	2	817.6
chp N×N	4	58	11	4	0	0	99.9
Clobber*	3	92	53	11	13	2	1198.6
Clobber	3	92	-	-	-	-	-
Colouring	3	44	38	13	0	9	188.7
1d	3	34	4	3	0	0	6.2
Arith	3	34	5	3	0	0	6.5
Find	3	50	3	4	0	0	13.8
Sort	3	59	20	10	3	0	540.9
Add	4	41	7	2	0	0	8.5
PrizeA1	5	49	61	5	1	0	1300.1

Table 1: Experimental results

placed in the first row, and black ones in the second row. A player moves by using one of his tokens to replace an adjacent token of the opposite colour. The one who has no moves to take loses the game. The strategy for player 2 is: if player 1 removes a token t in the nth column, where n is odd (resp. even), then player 2 removes the token in the $(n-1)$th (resp. $(n+1)$th) column to replace t.

Clobber*: The same as Clobber except that player 1 can only remove tokens in the odd columns. The strategy for player 2 is to replace any token that has just replaced her tokens.

Coloring: Two players take turns to paint blue or red on a 1xN grid. Painting red (resp. blue) on a cell is possible if the cell is not painted or painted with blue (resp. red). Painting on a coloured cell turns it into green. The game ends when all cells are colored. Player 1 wins if all cells are green. The strategy for player 2 is: If her opponent paints red (resp. blue) on some cell, then she paints blue (resp. red) on the same cell.

The following are programs in single-agent domains from [Mo *et al.*, 2016]:

1D: Visit all elements in an array from right to left.

Arith: Increase a variable from 0 to $2N$.

Find: Set the ith element of an array inb to true if the ith element of an integer array ina is not 0, and to false otherwise.

Sort: Sort an array by using a single loop.

Add: Increase a variable from 0 to AxB via a nested loop.

PrizeA1: Visit all the NxN cells through a nested loop.

Experimental results are summarized in Table 1. Here C is the number of states in the composite strategy, P is the number of generated predicates, U^- and U^+ denote the total times of calling $update^-$ and $update^+$, B and R denote the total times of backtracking and reset, respectively, and T (in seconds) is the total time for verification. Our system successfully verifies all the domains except Clobber. Note that we solve PrizeA1, which cannot be solved by [Mo *et al.*, 2016].

7 Related Works

General game playing [Genesereth *et al.*, 2005] aims at creating a system capable of playing arbitrary games, represented by the Game Description Language (GDL). Some works [Ramanujam and Simon, 2008; Zhang and Thielscher, 2015] based on PDL (Propositional Dynamic Logic) or GDL concerned about representing and reasoning about the internal structure of strategies. Xiong and Liu [2016] extended the situation calculus for strategy representation and reasoning, and used a simple fragment of Golog for representing structured strategies. However, these works do not consider the representation and automatic verification of general strategies.

There has been some progress on automated reasoning in the situation calculus. Claßen and Lakemeyer [2008] proposed a method to verify temporal properties of nonterminating Golog programs. Based on it, De Giacomo *et al.* [2010] devised techniques for verification of strategic properties of situation calculus game structures. However, it is unknown whether the fixed-point computation method used in these works can be efficiently implemented. Li and Liu [2015] and Mo *et al.* [2016] explored automatic verification of partial correctness of Golog programs via discovery of loop invariants. The idea is to start with a sufficient condition and iteratively strengthen it until it becomes an invariant and then check if it is necessary. So they strengthen existing formulas but do not update them when needed as we do. When a discovered invariant is not necessary, their verification simply fails, whereas we continue the search process. Moreover, they do not use counterexamples to guide the discovery process.

De Giacomo *et al.* [2016a; 2016b] investigated bounded action theories in the situation calculus. A bounded action theory is one which entails that, in every situation, the number of object tuples in the extension of fluents is bounded by a given constant. The verification of a first-order variant of the μ-calculus and the verification of ConGolog programs are decidable for such theories. However, in general, the action theories we consider in this paper are not bounded. For example, even the initial KB of our chomp action theory is not bounded, since there is no bound on the size of the grids.

8 Conclusions

In this paper, we have proposed FSA strategies to represent structured strategies that solve a class of (possibly infinitely many) games with similar structures, and formalized their semantics in the situation calculus. We have proposed a sound but incomplete method for verifying if an FSA strategy is a winning strategy by using counterexample-guided local search for sufficient and necessary invariants. This is a method for verifying safety properties of strategies in general. We implemented our method and experimented with both two-player and single-player domains, and our experiments yielded encouraging results. A limitation of our work is that we consider invariants of the form of a CNF formula where variables are implicitly universally quantified. In the future, we are interested in more expressive invariants which allow existential quantification. More importantly, based on the presented work, we will explore automatic synthesis of FSA strategies via counterexample-guided refinement.

Acknowledgments

We acknowledge support from the Natural Science Foundation of China under Grant No. 61572535.

References

[Alur *et al.*, 2002] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[Cermák *et al.*, 2014] Petr Cermák, Alessio Lomuscio, Fabio Mogavero, and Aniello Murano. MCMAS-SLK: A model checker for the verification of strategy logic specifications. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 525–532, 2014.

[Chatterjee *et al.*, 2010] Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. Strategy logic. *Inf. Comput.*, 208(6):677–693, 2010.

[Clarke *et al.*, 2000] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, 2000.

[Clarke *et al.*, 2003] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[Claßen and Lakemeyer, 2008] Jens Claßen and Gerhard Lakemeyer. A logic for non-terminating golog programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008, Sydney, Australia, September 16-19, 2008*, pages 589–599, 2008.

[De Giacomo *et al.*, 2010] Giuseppe De Giacomo, Yves Lespérance, and Adrian R. Pearce. Situation calculus based programs for representing and reasoning about game structures. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*, 2010.

[De Giacomo *et al.*, 2016a] Giuseppe De Giacomo, Yves Lespérance, and Fabio Patrizi. Bounded situation calculus action theories. *Artif. Intell.*, 237:172–203, 2016.

[De Giacomo *et al.*, 2016b] Giuseppe De Giacomo, Yves Lespérance, Fabio Patrizi, and Sebastian Sardiña. Verifying congolog programs on bounded situation calculus theories. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 950–956, 2016.

[de Moura and Bjørner, 2008] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-2008)*, pages 337–340, 2008.

[Ferguson, 2014] Thomas Ferguson. *Game Theory*. Mathematics Department, UCLA, 2014.

[Genesereth *et al.*, 2005] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[Hu and Levesque, 2011] Yuxiao Hu and Hector J. Levesque. A correctness result for reasoning about one-dimensional planning problems. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 2638–2643, 2011.

[Levesque *et al.*, 1997] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997.

[Levesque, 2005] Hector J. Levesque. Planning with loops. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 509–515, 2005.

[Li and Liu, 2015] Naiqi Li and Yongmei Liu. Automatic verification of partial correctness of golog programs. In *Proc. of IJCAI-15*, pages 3113–3119, 2015.

[Lomuscio *et al.*, 2009] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 682–688, 2009.

[Mo *et al.*, 2016] Peiming Mo, Naiqi Li, and Yongmei Liu. Automatic verification of golog programs via predicate abstraction. In *Proc. of the European Conference on Artificial Intelligence (ECAI-2016)*, 2016.

[Ramanujam and Simon, 2008] Ramaswamy Ramanujam and Sunil Easaw Simon. Dynamic logic on games with structured strategies. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008, Sydney, Australia, September 16-19, 2008*, pages 49–58, 2008.

[Reiter, 2001] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[Srivastava *et al.*, 2008] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. Learning generalized plans using abstract counting. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*, pages 991–997, 2008.

[Xiong and Liu, 2016] Liping Xiong and Yongmei Liu. Strategy representation and reasoning in the situation calculus. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, pages 982–990, 2016.

[Zhang and Thielscher, 2015] Dongmo Zhang and Michael Thielscher. A logic for reasoning about game strategies. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 1671–1677, 2015.