Selective Approaches for Solving Weak Games

Malte Helmert[1], Robert Mattmüller[1], and Sven Schewe[2]

[1] Albert-Ludwigs-Universität Freiburg
79110 Freiburg, Germany
{helmert|mattmuel}@informatik.uni-freiburg.de
[2] Universität des Saarlandes
66123 Saarbrücken, Germany
schewe@cs.uni-sb.de

Abstract. Model-checking alternating-time properties has recently attracted much interest in the verification of distributed protocols. While checking the validity of a specification in alternating-time temporal logic (ATL) against an *explicit* model is cheap (linear in the size of the formula and the model), the problem becomes EXPTIME-hard when *symbolic* models are considered. Practical ATL model-checking therefore often consumes too much computation time to be tractable.
In this paper, we describe a novel approach for ATL model-checking, which constructs an explicit weak model-checking game on-the-fly. This game is then evaluated using heuristic techniques inspired by efficient evaluation algorithms for and/or-trees.
To show the feasibility of our approach, we compare its performance to the ATL model-checking system MOCHA on some practical examples. Using very limited heuristic guidance, we achieve a significant speedup on these benchmarks.

1 Introduction

Alternating-time temporal logics like ATL [2] have recently attracted much interest in the multi-agent community [15, 16, 14, 17]. A typical application of alternating-time model-checking is the verification of distributed protocols. In the design of such protocols, we are often interested in the strategic abilities of certain agents (cf. [15, 16, 3]). For example, in a contract-signing protocol, it is important to ensure that while Alice and Bob can cooperate to sign a contract, Alice is never able to obtain Bob's signature unless Bob can also obtain Alice's signature, and vice versa. Such properties can be expressed in ATL, which extends the branching-time temporal logic CTL [7] with modalities that quantify over the strategic choices of groups of agents.

As in the case of CTL, the model-checking problem for ATL reduces to solving weak games [2]. Weak games are a particular simple version of parity games, where all vertices within a strongly connected component have the same color. ATL model-checking therefore seems to be simple: Given an alternating transition system $\mathcal{A}$ (i. e., an *explicit* model for ATL) and a specification φ, the size of the weak model-checking game is in $O(|\mathcal{A}| \cdot |\varphi|)$, where $|\mathcal{A}|$ denotes the

size of $\mathcal{A}$ and $|\varphi|$ the number of subformulas of φ. The resulting weak game can be solved in time linear in its size. It thus seems, at first glance, that ATL inherits the model-checking complexity from CTL. Indeed, MOCHA [3], the only available tool for ATL model-checking, generalizes a symbolic backward approach for CTL model-checking [5].

In light of these similarities, it might appear somewhat surprising that the performance of ATL model checking does not seem to meet the high standards set by CTL model checking. Kremer and Raskin, for example, observed exceptionally large time consumption (and, partly, abortions) when model checking simple properties of small protocols [15]. One possible explanation for this discrepancy is that, despite the identical model-checking complexity of $O(|\mathcal{A}|\cdot|\varphi|)$ for explicit models, the model-checking complexities of CTL and ATL do *not* coincide for symbolic models: while symbolic model-checking is PSPACE-complete for CTL, it becomes EXPTIME-complete for ATL, as recently shown by van der Hoek et al. [18]. In practice, model-checkers use succinct symbolic representations for models, such as RML for MOCHA [3] or PROMELA for SPIN [12], so that the symbolic model checking complexity is of paramount importance.

In addition to the increased complexity of ATL model-checking, there is a significant structural difference between model-checking ATL and CTL formulas. When we model-check a CTL formula, it is not unusual that the *complete* (reachable) state space needs to be explored (consider, e.g., a proof that φ holds during all computations, $\mathsf{AG}\varphi$). For many ATL formulas, there is no such necessity of complete exploration: to prove that a group A of agents can enforce that φ globally holds ($\langle\!\langle A\rangle\!\rangle\mathsf{G}\varphi$), we only need to consider a fragment of the states, defined by the strategies followed by these agents.

We therefore propose an approach that constructs the explicit model-checking game from a symbolic representation of the model-checking problem on-the-fly. Different from a forward-backward approach, we do not start by constructing the *complete* set of forward reachable states. Instead, we adopt heuristic best-first search methods for solving reachability games in and/or-trees to weak games, and finish a model-checking run as soon as we can prove that the considered set of states is sufficiently large for one of the players to have a winning strategy. Our adoption takes into account that, unlike and/or-trees, weak games can not only be won by a player by reaching winning states, but also by forcing the game to stay in vertices with a winning color. It turns out that selectively exploring the space of game vertices is a powerful method for obtaining small proof graphs and fast evaluation results.

Organization of the Paper. The following section introduces weak games, followed by Section 3 describing our approach for their solution. We then discuss the application of these techniques to ATL model-checking in Section 4. We close with a presentation (Section 5) and discussion (Section 6) of our results.

2 Weak Games

A *weak game* is a tuple $\mathcal{G} = \langle V_{even}, V_{odd}, E, v_0, \alpha\rangle$, where

- $V = V_{even} \uplus V_{odd}$ is a finite set of vertices, partitioned into V_{even} and V_{odd}, with a designated initial vertex $v_0 \in V$.
- $E \subseteq V \times V$ is a set of edges.
- $\alpha : V \to \mathbb{N}$ is a coloring function, satisfying $(v, w) \in E \Rightarrow \alpha(v) \leq \alpha(w)$.

Each vertex $v \in V$ has outdegree at least 1 in the directed graph (V, E). For a vertex $v \in V_{even}$ we say that *even* is the owner of v ($owner(v) = even$) and for a vertex $v \in V_{odd}$ we say that *odd* is the owner of v ($owner(v) = odd$). We say that the level of v is *even* ($level(v) = even$) iff $\alpha(v)$ is even and that that the level of v is *odd* ($level(v) = odd$) iff $\alpha(v)$ is odd. For each natural number $n \in \alpha(V)$ in the mapping of α, the vertices $\alpha^{-1}(n)$ colored with n are called a *level*.

The winning condition for weak games is defined in terms of runs. A *run* of a game is an infinite sequence $v_0 v_1 v_2 \ldots$ in V^ω such that $(v_i, v_{i+1}) \in E$ is an edge if v_{i+1} is a successor of v_i. A run is *winning* for player *even* (*odd*) iff the highest color of vertices occurring infinitely often in the run is even (odd). Due to the monotonicity condition for vertex colors, almost all vertices in a run have the same color, and every run is winning either for player *even* or *odd*.

Weak games are a special form of parity games, and consequently one player wins with a memoryless strategy [9]. A (memoryless) *strategy* for player $p \in \{odd, even\}$ is a mapping $s_p : V_p \to V$ such that $(v, v') \in E$ whenever $s_p(v) = v'$. A run $v_0 v_1 v_2 \ldots$ is in accordance with a strategy s_p iff, for all $i \in \mathbb{N}$, $v_i \in V_p \Rightarrow v_{i+1} = s_p(v_i)$ holds. A strategy s_p is *winning* for player p, iff all runs in accordance with s_p are winning for player p.

A vertex v is winning for player p iff she has a winning strategy in the game $\langle V_{even}, V_{odd}, E, v, \alpha \rangle$, and a game is *won* by player p iff the initial vertex is winning for her. *Solving a game* means determining by which player it is won.

3 Solving Weak Games

Weak games with n vertices and e edges can be solved in time $O(n+e)$ following a simple backward approach. For player $p \in \{even, odd\}$ and a given (partial) labeling of the game vertices as winning for *even* or winning for *odd*, define the *p-attractor* to be the minimal set V_p such that:

- a vertex $v \in V$ with $owner(v) = p$ belongs to V_p if some successor $w \in succ(v)$ is labeled as winning for p or belongs to V_p (player p can choose to play into a vertex winning for p), and
- a vertex $v \in V$ with $owner(v) \neq p$ belongs to V_p if each successor $w \in succ(v)$ is labeled as winning for p or belongs to V_p (the opponent of p is forced to play into a vertex winning for p).

The backward algorithm proceeds in phases, iterating until the initial vertex is labeled as winning for either player. In every phase, it considers the set of unlabeled vertices V_{max} whose color is maximal among all unlabeled vertices and labels the vertices in V_{max} as winning for p, where p is *even* (*odd*) if the

color of the vertices in V' is even (odd). It then computes the p-attractor V_p and labels the vertices in V_p as winning for p. The algorithm can easily be implemented in such a way that every vertex and edge is considered only once (cf. [8, 4]), proving the $O(n + e)$ complexity bound.

A disadvantage of this approach is that usually almost all vertices of the game need to be considered. On the other hand, only a small fragment of the state space is forward reachable in most model-checking games. An obvious improvement in such situations is to construct all forward reachable states in a first phase, and then solve the smaller resulting game using a standard backward algorithm. The complexity of this approach is linear in the size of the forward reachable sub-game.

For larger examples, this is still unsatisfactory: knowing a winning strategy beforehand, it suffices to consider only the fragment of the forward reachable vertices defined by this strategy. For example, in games corresponding to and/or-trees of uniform outdegree $b \geq 2$ and depth d, exploiting the knowledge of a winning strategy reduces the number of vertices that need to be considered from $O(b^d)$ to $O(b^{d/2})$ [13]. In other words, the number of vertices to consider is reduced to its square root.

This raises the question whether we can identify winning states without the need of completely exploring the game graph. This is obviously the case for vertices which are won because they belong to the attractor of a previously labeled set of vertices: If all successors of a vertex are winning for p, or if the vertex is owned by p and has at least one winning successor, then it is winning. However, it is also possible to define a winning criterion for vertices which are winning for a player because they belong to a level of that player and the opponent cannot force a run to leave this level without playing into a losing vertex. For this purpose, we define a *force-set* of player $p \in \{even, odd\}$ to be a set F of vertices in the same level $level(F) = \{p\}$ with the following properties:

- each vertex $v \in F$ with $owner(v) = p$ has some successor $w \in succ(v)$ which belongs to F or is already labeled as winning for p, and
- each vertex $v \in F$ with $owner(v) \neq p$ only has successors $w \in succ(v)$ which belong to F or are already labeled as winning for p.

Vertices in a force-set F of player p are winning for player p, following a strategy which maps vertices in F to vertices in F or to vertices labeled as winning for p.

3.1 A Strategic Forward-Backward Approach

Our algorithm for solving weak games incrementally constructs the game graph. Different from a forward-backward approach, we do not start by constructing the *complete* set of forward reachable states, but rather aim at an early (partial) evaluation of the constructed fragment.

The central data structure of the algorithm is the *partial game graph*, which represents a subgraph of the game graph (V, E) of the weak game to be solved. At any time during the execution of the algorithm, vertices in the partial game graph are partitioned into three groups:

```
Procedure ExpandFringeVertex(v: Vertex):
    change the status of v from "fringe" to "pending"
    for all outgoing edges (v, v') ∈ E:
        add (v, v') to the partial game graph
        if v' is unconstructed:
            add v' to the fringe
    if v has an evaluated successor v' ∈ succ(v) with winner(v') = owner(v):
        EvaluatePendingVertex(v, owner(v))
    else if all successors v' ∈ succ(v) are evaluated:
        EvaluatePendingVertex(v, opponent(owner(v)))
```

Fig. 1. Expanding a vertex moves it from the fringe to the set of pending vertices and add its outgoing edges to the graph, creating new fringe vertices where necessary. The new vertex is immediately evaluated if possible.

- An *evaluated* vertex v has already been classified as winning for *even* or winning for *odd*, and all outgoing edges $(v, w) \in E$ are represented in the partial game graph.
- A *pending* vertex v has not yet been classified, but all outgoing edges $(v, w) \in E$ are represented in the partial game graph.
- A *fringe* vertex v has not yet been classified, and none of its outgoing edges are is represented in the partial game graph.

Evaluated, pending and fringe vertices are called *constructed*, while vertices not represented in the partial game graph at all are called *unconstructed*.

The central primitive operations of the algorithm are *expanding a fringe vertex*, which transforms a fringe vertex into a pending vertex and adds its unconstructed successors to the fringe, and *evaluating a pending vertex*, which transforms a pending vertex into an evaluated vertex. Both operations can lead to the evaluation of further vertices. The overall *solving procedure* repeatedly expands vertices and identifies force-sets, triggering the ensuing vertex evaluations until the initial vertex of the game is evaluated. At this point, the algorithm stops. We now explain these three parts of the algorithm in sequence.

Expanding a Fringe Vertex. When a fringe vertex is expanded, it is removed from the fringe and becomes a pending vertex. Pending vertices must have their outgoing edges represented in the partial game graph, so they are added at this step, which may lead to the creation of new fringe vertices.

It may be the case that the winner for the expanded vertex can be determined immediately: If the owner of the vertex can play into a winning vertex, she wins the expanded vertex. Conversely, if the owner of the vertex is forced to play into a losing vertex for lack of other possibilities, she loses the expanded vertex. In either situation, procedure EvaluatePendingVertex is called to mark the vertex as evaluated and propagate the evaluation result upwards in the partial game graph where possible. The pseudo-code for the expansion procedure is depicted in Figure 1.

```
Procedure EvaluatePendingVertex(v: Vertex, p: Player):
    change the status of v from "pending" to "evaluated"
    set winner(v) to p
    for all pending predecessors v' ∈ pred(v):
        if owner(v') = p:
            EvaluatePendingVertex(v', p)
        else if v' has no unevaluated successors:
            EvaluatePendingVertex(v', p)
```

Fig. 2. Whenever a vertex is evaluated as winning for either player, the evaluation result is propagated up the partial game graph until no further evaluations are possible.

Evaluating a Pending Vertex. The evaluation procedure moves a vertex v from the set of pending vertices to the set of evaluated vertices and stores the winning player p in $\mathit{winner}(v)$.

Evaluating a vertex may lead to further winning vertices being found: If the partial game graph contains a pending predecessor v' of v which is owned by p, then p can choose to play into v from there and consequently also wins v'. A pending predecessor v' owned by the opponent of p is winning for p if all its successors are winning for p. In the evaluation procedure, it suffices to test that such a vertex v' has no pending or fringe successors; it cannot have evaluated successors won by the opponent of p, because in that case it would have been evaluated as winning for the opponent in an earlier call to the evaluation procedure.

In either case where a winning predecessor v' is found, the evaluation procedure is called recursively to mark v' as winning and propagate the evaluation result. The pseudo-code for the evaluation procedure is depicted in Figure 2.

Overall Solution Algorithm To solve a weak game, the overall solution algorithm starts with an empty game graph, which only contains the initial vertex as a fringe vertex.

It then proceeds iteratively by locating force-sets of pending vertices and evaluating the contained vertices as won by their owner, or if no force-set can be found, expanding a fringe vertex which can be selected with an arbitrary selection strategy. This process is repeated until the initial vertex is evaluated (pseudo-code in Figure 3).

The algorithm is guaranteed to terminate: In each iteration, either a force-set can be identified or a fringe vertex can be expanded. In particular, if there are no fringe vertices left, the complete reachable part of the game graph has been constructed, in which case the set of all pending vertices of maximal color forms a force-set. (If no pending vertices remain, the initial vertex must already be evaluated.) It is thus not possible for the overall search procedure to arrive in a situation where it is impossible to proceed further. It is clear that the algorithm must terminate after at most $2|V|$ iterations of the main loop, because each iteration either moves a vertex from the fringe to the set of pending vertices or from there to the set of evaluated vertices.

```
Procedure SolveWeakGame():
    initialize the sets of evaluated and pending vertices with ∅
    initialize the set of fringe vertices with {v_0}
    while v_0 is not evaluated:
        if we can locate a force-set F among the set of pending vertices:
            for all v ∈ F:
                EvaluatePendingVertex(v,level(v))
        else:
            pick a fringe vertex v
            ExpandFringeVertex(v)
```

Fig. 3. Starting from a partial game graph containing only the initial vertex, expand vertices and evaluate force-sets until the initial vertex is evaluated.

The remaining open question is how the algorithm locates force-sets. A complete – but expensive – method to identify force-sets is to continuously test if a force-set exists using a strategy similar to that used by the pure backward algorithm. However, the complexity of this approach is too high, scaling with the *product* of the size of the constructed sub-game and the maximal size of a single level.

Thus, the algorithm pursues the less ambitious approach of only searching for force-sets that consist of *all pending vertices within a given level*. Testing this property can be performed very efficiently, as we will now discuss. Although it cannot find all force-sets, it already provides good results (cf. Section 5).

Efficient Implementation. We assume that the basic set operations of adding an element, removing an element and testing membership can be performed in constant time. Hash tables with randomized hash functions can achieve this in the *expected* case. (If we do not want to resort to randomization, we can instead use AVL trees, in which case a logarithmic factor needs to be added to our complexity result.)

We also assume that it is possible to enumerate the set of successor vertices $succ(v)$ of a given vertex v in time linear in $|succ(v)|$.

Under these assumptions, procedure ExpandFringeVertex only requires time $O(|succ(v)|)$ for a given vertex v (excluding any time spent within EvaluatePendingVertex), and as it is called at most once for each vertex in the partial game graph, the total time spent in this procedure is $O(|E'|)$, where E' is the set of edges in the partial game graph upon termination.

To efficiently determine the pending predecessors of a vertex, we can maintain sets $pred(v)$ for all constructed vertices, adding each vertex v' to the predecessor set of all its successors as it is constructed. (We never need to refer to predecessors which are not part of the partial game graph.) To efficiently determine whether a vertex has unevaluated successors, we can keep track of the *number* of such successors for all vertices in the partial game graph. Maintaining the consistency of these numbers is easy to achieve without increasing the complexity of the search procedures. Excluding recursive invocations, procedure EvaluatePendingVertex

thus runs in time linear in the number of constructed predecessors of a given vertex v, again leading to an overall bound of $O(|E'|)$ because each vertex in the partial game graph is evaluated at most once.

To efficiently track whether the pending vertices of a given level form a force set, we maintain a single counter for each level which tracks the number of *violating* vertices in this level, and a set of levels for which this counter is currently 0. A pending vertex v violates the force-set condition iff it is owned by the level owner and has no pending successors in the same level or is owned by the other player and has a fringe successor or a pending successor with a higher color. (Note that we can ignore evaluated successors for testing the force-set condition because the propagation of evaluation results is already adequately taken care of by procedure EvaluatePendingVertex.) We thus only need to keep track of one additional number for each constructed vertex, which either counts the number of pending successors in the same level (for vertices v with $owner(v) = level(v)$), or the combined number of fringe successors and pending successors with a higher color (for other vertices). Again, keeping track of these numbers does not increase the asymptotical run-time of the algorithm.

If, finally, we also maintain a hash table which maps each color in the partial game graph to the corresponding set of pending vertices, procedure SolveWeakGame can be implemented in such a way that the overhead for each call to EvaluatePendingVertex or ExpandFringeVertex is constant, leading to the following result.

Theorem 1. *Procedure SolveWeakGame is a sound and complete algorithm for the problem of solving weak games. Its runtime is bounded by $O(|E'|)$, where E' is the set of edges constructed.*

The theorem follows from the previous discussion. In particular, termination and the run-time bound have already been been established, and for soundness, observe that vertices are only evaluated if they belong to a force-set or if their evaluation immediately follows from that of already evaluated successors.

4 Games and ATL

4.1 ATL

Alternating-time temporal logic (ATL) extends the classical computation tree logic (CTL) with path quantifiers $\langle\langle A \rangle\rangle$ and $[\![A]\!]$, expressing that a group A of agents has a strategy to accomplish a goal (defined by the respective path formula). For a definition of ATL formulas, we first introduce the structures over which a formula is interpreted. An alternating transition system (ATS) is a tuple

$$\mathcal{A} = \langle \Pi, \Sigma, Q, q_0, \pi, \delta \rangle,$$

consisting of a finite set Π of atomic propositions, a finite set Σ of agents, a finite set Q of states with a designated initial state q_0, a labeling function

$\pi : Q \to 2^{\Pi}$ that decorates each state with a subset of the atomic propositions, and a transition function $\delta : Q \times \Sigma \to 2^{2^Q}$. Intuitively, δ maps a state q and an agent a to the choices available to a at q. For any state $q \in Q$ and set of agents $A \subseteq \Sigma$, we define the set of *joint decisions* $\Delta(q, A)$ of A in state q as $\Delta(q, A) = \{ \bigcap_{a \in A} Q_a \mid Q_a \in \delta(q, a) \text{ for all } a \in A \}$. Once all agents $a \in \Sigma$ have made their choice $Q_a \in \delta(q, a)$ in a state q, the successor state must be uniquely determined. We thus require δ to be defined such that, in any state q, all joint decisions in $\Delta(q, \Sigma)$ are singletons.

ATL formulas are interpreted over an alternating transition system $\mathcal{A} = \langle \Pi, \Sigma, Q, q_0, \pi, \delta \rangle$. An ATL formula can be formed using the following grammar:

$$\varphi \ ::= \ \mathit{true} \mid \mathit{false} \mid p \mid \neg p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\!\langle A \rangle\!\rangle \bigcirc \varphi \mid [\![A]\!] \bigcirc \varphi \mid \\ \langle\!\langle A \rangle\!\rangle \varphi \mathsf{U} \varphi \mid [\![A]\!] \varphi \mathsf{U} \varphi \mid \langle\!\langle A \rangle\!\rangle \varphi \mathsf{W} \varphi \mid [\![A]\!] \varphi \mathsf{W} \varphi,$$

where $p \in \Pi$ is an atomic proposition, and $A \subseteq \Sigma$ is a set of agents. (Note that this definition deviates slightly from the original definition of ATL. The variant we use is strictly more expressive; e. g., in the original definition of ATL $\langle\!\langle A \rangle\!\rangle \varphi \mathsf{W} \psi$ cannot be expressed.) Intuitively, a formula $\langle\!\langle A \rangle\!\rangle \tau$ expresses the capability of the agents in A to enforce the path formula τ if they always have to make their choices before the other agents, while $[\![A]\!] \tau$ is the weaker requirement expressing that the agents in A can enforce the path formula τ if they only need to fix their decisions after their opponents made their choices.

For an ATL formula φ with atomic propositions Π and an alternating transition system $\mathcal{A} = \langle \Pi, \Sigma, Q, q_0, \pi, \delta \rangle$, $\|\varphi\|_{\mathcal{A}} \subseteq Q$ denotes the set of states where φ holds. The set $\|\varphi\|_{\mathcal{A}}$ is defined inductively along the structure of φ:

- Atomic propositions are interpreted as follows: $\|\mathit{true}\|_{\mathcal{A}} = Q$, $\|\mathit{false}\|_{\mathcal{A}} = \emptyset$, $\|p\|_{\mathcal{A}} = \{ q \in Q \mid p \in \pi(q) \}$ and $\|\neg p\|_{\mathcal{A}} = \{ q \in Q \mid p \notin \pi(q) \}$.
- As usual, conjunction and disjunction are interpreted as intersection and union, respectively: $\|\varphi \wedge \psi\|_{\mathcal{A}} = \|\varphi\|_{\mathcal{A}} \cap \|\psi\|_{\mathcal{A}}$ and $\|\varphi \vee \psi\|_{\mathcal{A}} = \|\varphi\|_{\mathcal{A}} \cup \|\psi\|_{\mathcal{A}}$.
- A state $q \in Q$ is in $\|\langle\!\langle A \rangle\!\rangle \bigcirc \varphi\|_{\mathcal{A}}$ if the agents A can make a joint decision $Q_A \in \Delta(q, A)$ such that, for all joint decisions $Q_{\Sigma \setminus A} \in \Delta(q, \Sigma \setminus A)$ of the other agents, φ holds in the successor state ($Q_A \cap Q_{\Sigma \setminus A} \subseteq \|\varphi\|_{\mathcal{A}}$).
- A state $q \in Q$ is in $\|[\![A]\!] \bigcirc \varphi\|_{\mathcal{A}}$ if for all joint decisions $Q_{\Sigma \setminus A} \in \Delta(q, \Sigma \setminus A)$ of the other agents, the agents A can make a joint decision $Q_A \in \Delta(q, A)$ such that φ holds in the successor state ($Q_A \cap Q_{\Sigma \setminus A} \subseteq \|\varphi\|_{\mathcal{A}}$).
- The remaining temporal operators are defined as fixed points.
  - $\|\langle\!\langle A \rangle\!\rangle \varphi \mathsf{U} \psi\|_{\mathcal{A}}$ ($\|\langle\!\langle A \rangle\!\rangle \varphi \mathsf{W} \psi\|_{\mathcal{A}}$) is the smallest (greatest) set X satisfying $\|\psi\|_{\mathcal{A}} \subseteq X \subseteq \|\varphi \vee \psi\|_{\mathcal{A}}$ with the following property:
    For all $q \in X \setminus \|\psi\|_{\mathcal{A}}$, the agents in A can make a joint decision $Q_A \in \Delta(q, A)$ such that, for all joint decisions $Q_{\Sigma \setminus A} \in \Delta(q, \Sigma \setminus A)$ of the other agents, the successor state is in X ($Q_A \cap Q_{\Sigma \setminus A} \subseteq X$), and
  - $\|[\![A]\!] \varphi \mathsf{U} \psi\|_{\mathcal{A}}$ ($\|[\![A]\!] \varphi \mathsf{W} \psi\|_{\mathcal{A}}$) is the smallest (greatest) set X satisfying $\|\psi\|_{\mathcal{A}} \subseteq X \subseteq \|\varphi \vee \psi\|_{\mathcal{A}}$ with the following property:
    For all $q \in X \setminus \|\psi\|_{\mathcal{A}}$ and all joint decisions $Q_{\Sigma \setminus A} \in \Delta(q, \Sigma \setminus A)$ of the other agents, the agents A can make a joint decision $Q_A \in \Delta(q, A)$ such that the successor state is in X ($Q_A \cap Q_{\Sigma \setminus A} \subseteq X$).

$\mathcal{A}$ is a *model* of a specification φ iff φ holds in the initial state ($q_0 \in \|\varphi\|_{\mathcal{A}}$).

4.2 Weak Games for ATL Model-Checking

Given an ATS $\mathcal{A}$ and an ATL formula φ, model-checking $\mathcal{A}$ naturally reduces to solving a weak model-checking game $\mathcal{G}^{\varphi}_{\mathcal{A}}$. The vertices of this game essentially consist of pairs of states of $\mathcal{A}$ and subformulas of φ.

Constructing the Game Graph. Intuitively, an ATL model-checking game is concurrently played on the formula tree and on the alternating transition system. It is technically more convenient to identify an until or wait-for formula $\psi = (\!(A)\!)\psi' \mathsf{V} \psi''$ ($(\!(A)\!) \in \{\langle\!\langle A \rangle\!\rangle, [\![A]\!]\}, \mathsf{V} \in \{\mathsf{U}, \mathsf{W}\}$) with the equivalent formula $\psi'' \vee \psi' \wedge (\!(A)\!) \bigcirc (\!(A)\!)\psi' \mathsf{V} \psi''$. The extended set Φ of subformulas of a formula φ thus consists of the following formulas:

- each subformula ψ of φ,
- for each subformula $\psi = \langle\!\langle A \rangle\!\rangle \psi' \mathsf{U} \psi''$ or $\psi = \langle\!\langle A \rangle\!\rangle \psi' \mathsf{W} \psi''$ of φ, the formulas $\overline{\psi} = \langle\!\langle A \rangle\!\rangle \bigcirc \psi$ and $\widehat{\psi} = \psi' \wedge \langle\!\langle A \rangle\!\rangle \bigcirc \psi$, and
- for each subformula $\psi = [\![A]\!] \psi' \mathsf{U} \psi''$ or $\psi = [\![A]\!] \psi' \mathsf{W} \psi''$ of φ, the formulas $\overline{\psi} = [\![A]\!] \bigcirc \psi$ and $\widehat{\psi} = \psi' \wedge [\![A]\!] \bigcirc \psi$.

The formulas ψ, $\overline{\psi}$ and $\widehat{\psi}$ are called *connected* (with the intuition that they form a strongly connected component in a subformula graph), and formulas of the form $\langle\!\langle A \rangle\!\rangle \bigcirc \psi$ and $[\![A]\!] \bigcirc \psi$ are called *temporal*.

The model checking game has two types of vertices:

- For each state $q \in Q$ of the model $\mathcal{A}$ and formula $\psi \in \Phi$ in the extended set of subformulas of φ, there is a *full-move* vertex (q, ψ), representing the situation where q is the current state in the model and formula ψ must be proved.
- For each *temporal* formula $\langle\!\langle A \rangle\!\rangle \bigcirc \psi \in \Phi$ or $[\![\Sigma \setminus A]\!] \bigcirc \psi \in \Phi$ in the extended set of subformulas of φ, state $q \in Q$ and joint decision $Q' \in \Delta(q, A)$, there is a *half-move* vertex (q, ψ, A, Q'), representing the situation where q is the current state in the model, formula ψ must be proved, and the agents in A have already made their next joint decision Q'.

It is computationally more convenient to use a variant of weak games where some vertices, namely those which refer to literals, have no successors, but are evaluated immediately. Such vertices appear as sinks in the game graph.

The weak model-checking game has the following transitions:

- There is a transition from (q, ψ) to (q, ψ') if ψ' is a direct subformula of ψ, where until and wait-for formulas $(\!(A)\!)\psi \mathsf{V} \psi'$ are again interpreted as disjunctions $\psi' \vee \psi \wedge (\!(A)\!) \bigcirc (\!(A)\!)\psi \mathsf{V} \psi'$.
- There is a transition from (q, ψ) to (q, ψ, A, Q') if ψ is a temporal formula.
- There is a transition from (q, ψ, A, Q') to (q', ψ') if $q' \in Q' \cap Q_{\Sigma \setminus A}$ for some joint decision $Q_{\Sigma \setminus A} \in \Delta(q, \Sigma \setminus A)$ of the agents not in A, and $\psi = (\!(A)\!) \bigcirc \psi'$.

Game Construction. To construct a weak game $\mathcal{G}^{\varphi}_{\mathcal{A}} = \langle V_{even}, V_{odd}, E, v_0, \alpha \rangle$, we only need to partition the set V of vertices into two sets V_{even} and V_{odd} of vertices, owned by the two players *even* and *odd*, find a suitable coloring function, and define the initial vertex. We assume that the objective of *even* is to prove that the model satisfies the formula, while the objective of *odd* is to disprove this.

The initial vertex is given by the pair (q_0, φ) consisting of the initial state q_0 of the model and the formula φ to be checked. A proper partition of V follows from the ATL semantics: Vertices (q, ψ) of the model-checking game whose formula part ψ is a conjunction or a temporal formula of the form $[\![A]\!] \bigcirc \psi'$, and vertices (q, ψ, A, Q') whose formula part ψ is a temporal formula of the form $\langle\!\langle A \rangle\!\rangle \bigcirc \psi'$ are owned by player *odd*; the remaining vertices are owned by player *even*. A proper coloring function maps a state $(q, \langle\!\langle A \rangle\!\rangle \psi \mathsf{U} \psi')$ or $(q, [\![A]\!] \psi \mathsf{U} \psi')$ to an odd color, and a state $(q, \langle\!\langle A \rangle\!\rangle \psi \mathsf{W} \psi')$ or $(q, [\![A]\!] \psi \mathsf{W} \psi')$ to an even color.

While the algorithm is sound and complete for every proper coloring function, the chosen coloring function can have a significant impact on the performance of the algorithm introduced in Section 3. The "standard" coloring is designed to create a *small* number of colors, which depend only on the formula. While this is convenient in a pure backwards analysis, it makes finding force-sets more difficult. In an optimal setting each strongly connected component of the game graph has a color of its own. While partitioning the game graph into strongly connected components is not cheaper than a complete evaluation, significant information can often be drawn from the *symbolic* representation of an alternating transition system.

A simple analysis of an RML specification suffices to identify counters that are only counted up (or down) and flags that are only set (or reset). Such situations naturally arise, e. g., in the definition of protocols. This allows for a simple construction of a ranking function γ on the abstract states, which is preserved by the concretization. Using such a ranking function results in a significant reduction of the size of levels, and therefore accelerates model-checking (cf. Section 5).

To achieve small levels, we create a coloring function which assigns the same color to two states (q, ψ) and (q', ψ') if and only if q and q' have the same rank and ψ and ψ' are equivalent or connected. In the protocol benchmark discussed in Section 5, this increases the number of colors from 2 colors in the standard coloring to about $5.3 \cdot 10^{33}$ colors, leaving the single levels in an accessible size.

5 Benchmarks and Results

To evaluate our algorithm, we implemented it in Java and tested it on some ATL properties of the Garay and MacKenzie multi-party contract signing protocol [10], using the RML formalization by Chadha et al. [6].

In particular, we considered the case of five agents (four contract-signing parties $P_1, \ldots, P_4$ and a trusted third party T) and the property of *protocol fairness*: A protocol is fair for an agent P_i following the protocol iff, whenever some other agent P_j obtains the signature of P_i, then P_i can obtain the signatures

of the other agents, even if they are all dishonest (i. e., do not follow the protocol) and do not cooperate. In ATL, we can express this property as

$$\mathsf{AG}\Big(\Big(\bigvee_{j\neq i} has\_sig(P_j, P_i)\Big) \rightarrow \langle\langle P_i\rangle\rangle \mathsf{F} \bigwedge_{j\neq i} has\_sig(P_i, P_j)\Big)$$

(The common G and F modalities can be expressed in ATL using W and U in the usual way [5]. The A path quantifier is synonymous with $\langle\langle\emptyset\rangle\rangle$.) Because the Garay and MacKenzie protocol is asymmetric, protocol fairness must be proved or disproved separately for each of the four contract-signing parties P_i.

As we observed in the introduction, selective explicit-state methods are only useful when checking properties which *can* be verified or refuted without considering the complete reachable state space. Given that the protocol fairness property is of the form $\mathsf{AG}\varphi$, *verifying* it requires constructing all reachable states. However, as originally shown by Chadha et al. [6], fairness is *violated* in the Garay and MacKenzie protocol for the case of $i \neq 4$, and thus in this case the property can serve as a useful benchmark for selective methods. The protocol is fair for agent P_4, so selective algorithms do not work well in this case.

We have model-checked the protocol fairness property for each agent using three different approaches:

- First, we used the MOCHA model checker, which solves the weak game corresponding to an ATL formula by a symbolic backward computation.
- Second, we implemented a standard explicit-state forward-searching evaluation strategy, exploring the game graph in depth-first order.
- Third, we considered our strategic forward-backward approach. As a selection strategy for choosing the next fringe vertex to expand, we employed a variant of the proof-number search algorithm used for evaluating and/or-trees [1].

A symbolic (non-strategic) forward-backward algorithm would have been a good fourth candidate approach, but it appears that no efficient implementation of such an algorithm is available. To at least compute a lower bound on the performance of such an algorithm, we performed a complete symbolic forward exploration – the first stage of a symbolic forward-backward algorithm – of the game graph using MOCHA's symbolic forward exploration capabilities (which are distinct from its ATL model-checking algorithm and can only be used to model-check invariants).

To initialize proof numbers for fringe vertices within our strategic forward-backward approach, we used the FF heuristic [11] with a *problem-dependent* goal formula, i. e., for each of the three properties we specified a collection of literals that we considered to be likely to be satisfied near "interesting" vertices in the game graph, which biases the exploration towards such vertices. Using such problem-dependent heuristics of course means that this is merely a *semi-automatic* approach: while the algorithm is sound and complete for all possible heuristics, a reasonable choice of heuristics is important for good performance.

	MOCHA	forward	SFB	strategic
fairness for P_1	21:15:07	failure	> 04:19:52	00:01:22
fairness for P_2	failure	failure	> 02:41:45	00:01:46
fairness for P_3	10:57:07	failure	> 06:25:33	00:01:26
fairness for P_4	00:39:14	failure	> 10:44:13	failure

Fig. 4. Run-time results for the protocol-fairness property. The four algorithms considered are MOCHA, explicit forward search, symbolic forward-backward search (SFB; only forward exploration counted), and strategic forward-backward search. Time is measured in hours, minutes and seconds (hh:mm:ss).

	evaluated	pending	fringe	reachable
fairness for P_1	130	934	37036	$2.4 \cdot 10^{14}$
fairness for P_2	298	2098	51814	$7.6 \cdot 10^{15}$
fairness for P_3	628	4765	31952	$7.4 \cdot 10^{17}$

Fig. 5. Numbers of evaluated, pending and fringe vertices generated by the strategic forward-backward algorithm. The total number of forward-reachable vertices is shown for comparison.

Using hand-tuned heuristic information is sufficient for the purposes of this investigation, in which our objective is to demonstrate the usefulness of selective game-solving approaches in general rather than the development of game-solving heuristics; however, the latter certainly remains as an important open problem.

The results of our experiment are shown in Figure 4.[3] We clearly see that selectivity pays off on this suite of benchmarks. Simple-minded explicit search methods like standard forward search cannot cope with this state space at all, and exhaustive symbolic methods require many hours of solution time where the selective approach terminates within a few minutes. In particular, it dramatically improves on a symbolic forward-backward exploration, which would require several hours for computing the set of forward reachable states alone. All this only applies to agents P_1, P_2 and P_3, however. For agent P_4, the complete reachable state space must be considered to prove protocol fairness, and there is no hope of achieving this with an explicit-state method.

Compared to traditional approaches, one advantage of our algorithm is that proofs (or refutations) of the checked properties are generated as part of the search. In particular, the subgraph of the partial game graph induced by the set of evaluated vertices forms an explicit proof of the property. In comparison, MOCHA only reports whether a given ATL formula holds or does not hold in a model, without providing further information. Figure 5 offers some statistics on the size of the explored state spaces, showing that the generated proofs are indeed very selective.

[3] Experiments were conducted on a standard Linux PC with a 3 GHz CPU and using a heap limit of 512 MB. All failures are due to running out of memory.

6 Discussion

We have presented a new algorithm for solving weak games, such as those arising from ATL model-checking problems, which is based on the idea of selectively generating only those parts of the game graph which are relevant to proving or disproving the hypothesized property. Using a combination of forward exploration to extend a partially constructed game graph and backward propagation of evaluation results, including the efficient detection of *force-sets* for situations where a player can force a run of the game to stay in a given level of the game graph indefinitely, the algorithm can solve a weak game in time which is linear in the *size of the subgame considered*, rather than linear in the size of the game. In the best case, this can lead to dramatic speedups compared to exhaustive approaches. In the worst case, the algorithm is still asymptotically optimal.

One significant advantage of the selective search method we present is that, unlike traditional methods for solving weak games symbolically, it generates a verifiable proof of the model-checking result. What is more, due to the selective nature of the search, we can expect such proofs to be comparatively small, because they are explicitly represented within the algorithm as subgraphs of the partial game graph that serves as a main data structure.

Of course, the flip side of this advantage is that selective search techniques only make sense for games where short proofs exist, i. e., where one player can force the game to remain in a comparatively small fragment of the overall game graph. If the complete game graph needs to be explored, there is little point in using a selective method, and one can expect better performance from a systematic symbolic algorithm.

Future Work. In the current form, the algorithm only finds force-sets that cover all pending vertices within a level. One might consider strengthening this approach by initiating an exhaustive evaluation after each expansion, but this is too expensive to be pursued. An interesting alternative is to use approximative methods based on the weak or strong connectivity structure of the pending vertices in a level. Both structures provide useful information since finding a force-set within a strongly or weakly connected component coincides with finding a force-set in a level.

Weak connectedness, in particular, can be efficiently tracked using classical union-find data structures. Thus, distinguishing weakly connected components within a level promises a good trade-off between (expensive) continuous re-evaluations and the coarse approximation of the basic method. Using this method, worst-case runtime is still quasi-linear in the number $|E'|$ of edges constructed by the algorithm, while the constructed subgraph is potentially smaller.

Another method to speed up the detection of force-sets is to use *thresholding forward-backward search*, where a *complete* backward evaluation is initiated after c, c^2, c^3, ... steps (for some constant $c > 1$). On a finer granularity, one could initiate the evaluation of a *level* after c, c^2, c^3, ... vertices of the level have been constructed. One strength of such a thresholding approach is that it retains the asymptotically optimal behavior of the basic algorithm.

Acknowledgement

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

References

1. L. V. Allis, M. van der Meulen, and H. J. van der Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.
2. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.
3. R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In *Proc. CAV*, pages 521–525, 1998.
4. H. R. Andersen. Model checking and boolean graphs. *Theor. Comput. Sci.*, 126(1):3–30, 1994.
5. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
6. R. Chadha, S. Kremer, and A. Scedrov. Analysis of multi-party contract signing. Technical Report 516, Université Libre de Bruxelles, 2004.
7. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, pages 52–71, 1981.
8. R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal μ-calculus. In *Proc. CAV '91*, pages 48–58, 1992.
9. E. A. Emerson and C. S. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. FOCS*, pages 368–377, 1991.
10. J. A. Garay and P. D. MacKenzie. Abuse-free multi-party contract signing. In *International Symposium on Distributed Computing*, volume 1693 of *LNCS*, pages 151–165, 1999.
11. J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
12. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
13. D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
14. S. Kremer. *Formal Analysis of Optimistic Fair Exchange Protocols*. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium, Dec. 2003.
15. S. Kremer and J.-F. Raskin. A game-based verification of non-repudiation and fair exchange protocols. *Journal of Computer Security*, 11(3):399–430, 2003.
16. A. Mahimkar and V. Shmatikov. Game-based analysis of denial-of-service prevention protocols. In *IEEE Computer Security Foundations Workshop*, pages 287–301, 2005.
17. M. Ryan and P.-Y. Schobbens. Agents and roles: Refinement in alternating-time temporal logic. In *Proc. ATAL*, pages 100–114, 2001.
18. W. van der Hoek, A. Lomuscio, and M. Wooldridge. On the complexity of practical ATL model checking. In *Proc. AAMAS*, 2006.