On the Complexity of Practical ATL Model Checking

Wiebe van der Hoek
Dept of Computer Science
University of Liverpool
Liverpool L69 7ZF, UK
wiebe@csc.liv.ac.uk

Alessio Lomuscio
Dept of Computer Science
University College London
London WC1E 6BT, UK
A.Lomuscio@cs.ucl.ac.uk

Michael Wooldridge
Dept of Computer Science
University of Liverpool
Liverpool L69 7ZF, UK
mjw@csc.liv.ac.uk

ABSTRACT

We investigate the computational complexity of reasoning about multi-agent systems using the cooperation logic ATL of Alur, Henzinger, and Kupferman. It is known that satisfiability checking is EXPTIME-complete for "full" ATL, and PSPACE-complete (in the general case) for the fragment of ATL corresponding to Pauly's Coalition Logic. In contrast, the model checking problems for ATL and Coalition Logic can both be solved in time polynomial in the size of the formula and the size of model against which the formula is to be checked. However, these latter results assume an *extensive* representation of models, in which all states of a model are explicitly enumerated. Such representations are not feasible in practice. In this paper we investigate the complexity of the ATL and Coalition Logic model checking problems for a more "reasonable" model representation known as SRML ("Simple Reactive Modules Language"), a simplified version of the actual model representation languages used for model checkers such as SMV and MOCHA. While it is unsurprising that, when measured against such representations, the model checking problems for ATL and Coalition Logic have a higher complexity than when measured against explicit state representations, we show that in fact the ATL and Coalition Logic model checking problems for SRML models have *the same complexity as the corresponding satisfiability problems*. That is, model checking ATL formulae against SRML models is EXPTIME-complete, and model checking Coalition Logic formulae against SRML models is PSPACE-complete. We conclude by investigating some technical issues around these results, and discussing their implications.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent Systems; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*; F.2 [**Analysis of algorithms and problem complexity**]

General Terms

Theory

AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.

Keywords

logic, verification, cooperation, complexity

1. INTRODUCTION

Cooperation logics – such as Pauly's Coalition Logic [19, 20, 9] and its temporal extension, the Alternating-time Temporal Logic (ATL) of Alur, Henzinger, and Kupferman [2] – have attracted much interest from the multi-agent systems community [23, 14, 15, 11]. Using such logics, it is possible to reason about the strategic powers of agents and coalitions of agents in game-like multi-agent scenarios.

If one aims to use logics such as ATL and Coalition Logic to reason about systems, then it is obvious to ask how complex the associated reasoning problems are for such logics. Given a particular (multi-agent) system ξ, there are essentially two approaches that may be adopted to reasoning about ξ: theorem proving (via satisfiability checking) and model checking [4]. The satisfiability problem for ATL is EXPTIME-complete, and is hence prohibitively expensive [7, 25]; satisfiability for Coalition Logic is "only" PSPACE-complete [19, 20, 21], and thus apparently easier, but of course this "easiness" is relative: PSPACE-completeness is an extremely negative complexity result!

In contrast, model checking approaches to reasoning about systems appear to be computationally easier. Model checking was originally developed as a technique for verifying that finite state concurrent systems satisfy specifications expressed in the language of temporal logic [8]. The basic idea is that the state transition graph of a system ξ can be understood as a model M_ξ for a temporal logic such as Computation Tree Logic (CTL): if we express desirable or undesirable properties of ξ as formulae φ of CTL, we can then check whether or not ξ has these properties by evaluating whether or not $M_\xi \models \varphi$, and this can be done in deterministic polynomial time [8, p.1044].

Now, one of the attractive features of ATL (and Coalition Logic) is that its model checking problem is apparently no harder than that of its ancestor CTL. More formally, the model checking problem, as discussed in [2] is as follows. We are given an ATS (i.e., an ATL model) S, a state q from S, a formula φ over S and we are asked whether $S, q \models \varphi$. It is shown in [2] that this problem is tractable: it can be solved in time $O(|S| \times |\varphi|)$, where $|S|$ is the size of S and $|\varphi|$ is the size of φ. However, this apparently positive result has a well-known catch. It assumes that the ATS S is *explicitly enumerated* in the input to the problem, and in particular, that all the states of S are listed in the input. Of course, if S has n Boolean variables, then there may be 2^n possibles states. Moreover, in a very recent result, Jamroga and Dix point out that this tractability result only goes through if the number of agents is *fixed*, and *not* considered a parameter of the problem [13]. They show that if

the number of agents is considered an input to the model checking problem, then model checking is NP-complete with respect to a representation of ATL models as *alternating transition systems*, and even worse – Σ_2^p-complete – for the representation of ATL models as *concurrent game structures*.

So, the tractability result for ATL model checking is perhaps misleading, since we have tractability *only* if we explicitly enumerate all states of the system in the input, and *only* if we assume the number of agents is fixed, and not part of the input.

The complexity of ATL model checking is thus particularly sensitive to the *representation* of the model given as input. An obvious question to ask is what kinds of representations are used *in practice*, by actual model checking systems. In fact, practical model checkers such as SMV (for CTL) [17], SPIN (for LTL) [12], and MOCHA (for ATL) [3] use a succinct *model specification language* for defining models. While the model specification languages used by these systems differ on details, they all have the flavour of programming languages, providing the verifier with a relatively high-level way of defining models. But this then raises the question of the extent to which the apparently positive complexity results for CTL, ATL, and Coalition Logic model checking hold when using such model representation languages. It comes as no surprise that model checking against such model representations will be harder. For example, a PSPACE-completeness result for CTL model checking against a representation of concurrent programs was obtained by Kupferman et al [16], and extended to combinations of CTL with knowledge in [22].

However, to date, the issue of complexity of ATL model checking for realistic model specification languages has not been investigated, and this is the aim of the present paper. The particular model specification language we focus on is called SRML ("Simple Reactive Modules Language"), and is a slightly simplified version of the REACTIVE MODULES LANGUAGE (RML) [1]. We are particularly interested in RML for several reasons: first, RML is the language that is actually used by the ATL model checking system MOCHA [3]; it has been shown to be well suited to building models of a range of multi-agent protocols; and RML programs can easily be used to code basic reactive agents. As we pointed out above, it is unsurprising that, assuming representations such as SRML, the model checking problems for ATL and Coalition Logic have a higher complexity than when measured against explicit state representations, particularly given the results of Jamroga and Dix [13]. However, we show that in fact the ATL and Coalition Logic model checking problems for SRML programs have *the same complexity as the corresponding satisfiability problems*! That is, model checking ATL formulae against SRML programs is EXPTIME-complete, and model checking Coalition Logic formulae against SRML programs is PSPACE-complete. There is thus a *huge* gulf – much larger than perhaps might be expected – between the provably tractable model checking problem for explicit state models and the provably *in*tractable model checking problem for "realistic" model representations. Indeed, it again raises the question of the relative merits of theorem proving *versus* model checking [10]. We begin by describing ATL and then SRML, the simplified variant of RML that we use throughout the paper. We present our main results in section 4. We conclude by investigating some technical issues around these results, and discussing their implications.

2. ATL

Alternating-time Temporal Logic (ATL) [2] can be understood as a generalisation of the well-known branching time temporal logic CTL [8], in which path quantifiers are replaced by *cooperation modalities*. A cooperation modality $\langle\!\langle C \rangle\!\rangle \varphi$, where C is a group of agents, expresses the fact that there exists a strategy profile for C such that by following this strategy profile, C can ensure φ. Thus, for example, the system requirement "agents 1 and 2 can cooperate to ensure that the system never enters a fail state" may be captured by the ATL formula $\langle\!\langle 1,2 \rangle\!\rangle \Box \neg fail$. The "$\Box$" temporal operator means "now and forever more": additional temporal connectives in ATL are "$\Diamond$" ("either now or at some point in the future"), "$\mathcal{U}$" ("until"), and "$\bigcirc$" ("in the next state").

To give a precise definition of ATL, we must first introduce the semantic structures over which formulae of ATL are interpreted. An *alternating transition system* (ATS) is a 6-tuple

$$S = \langle \Pi, \Sigma, Q, Q_0, \pi, \delta \rangle, \text{ where:}$$

- Π is a finite, non-empty set of *Boolean variables*;
- $\Sigma = \{a_1, \ldots, a_n\}$ is a finite, non-empty set of *agents*;
- Q is a finite, non-empty set of *states*, with $Q_0 \subseteq Q$ representing the *initial* states of Q;
- $\pi : Q \to 2^{\Pi}$ gives the set of Boolean variables satisfied in each state;
- $\delta : Q \times \Sigma \to 2^{2^Q}$ is the system transition function, which maps states and agents to the choices available to these agents. Thus $\delta(q, a)$ is the set of choices available to agent a when the system is in state q. We require that this function satisfy the requirement that for every state $q \in Q$ and every set $Q_1, \ldots, Q_n$ of choices $Q_i \in \delta(q, a_i)$, the intersection $Q_1 \cap \cdots \cap Q_n$ is a singleton.

(Initial states Q_0 were not considered part of ATSs in the original publications on ATL: we introduce them to be consistent with the behaviour of model checkers for ATL, which check formulae against the initial state of the system: see below.)

An ATL formula, formed with respect to an alternating transition system $S = \langle \Pi, \Sigma, Q, Q_0, \pi, \delta \rangle$, is then defined by the following grammar:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\!\langle C \rangle\!\rangle \bigcirc \varphi \mid \langle\!\langle C \rangle\!\rangle \Box \varphi \mid \langle\!\langle C \rangle\!\rangle \varphi \,\mathcal{U}\, \varphi$$

where $p \in \Pi$ is a Boolean variable, and $C \subseteq \Sigma$ is a set of agents. We assume the remaining connectives of classical logic ("$\bot$", "$\rightarrow$", "$\leftarrow$", "$\leftrightarrow$", "$\wedge$") are defined as abbreviations in the usual way, and define $\langle\!\langle C \rangle\!\rangle \Diamond \varphi$ as $\langle\!\langle C \rangle\!\rangle \top \,\mathcal{U}\, \varphi$.

To give the semantics of ATL, we need some further definitions. For two states $q, q' \in Q$ and an agent $a \in \Sigma$, we say that state q' is an *a-successor* of q if there exists a set $Q' \in \delta(q, a)$ such that $q' \in Q'$. Intuitively, if q' is an a-successor of q, then q' is a possible outcome of one of the choices available to a when the system is in state q. We denote by $succ(q, a)$ the set of a-successors to state q, and say that q' is simply a *successor* of q if for all agents $a \in \Sigma$, we have $q' \in succ(q, a)$; intuitively, if q' is a successor to q, then when the system is in state q, the agents Σ can cooperate to ensure that q' is the next state the system enters.

A *computation* of an ATS $\langle \Pi, \Sigma, Q, Q_0, \pi, \delta \rangle$ is an infinite sequence of states $\lambda = q_0, q_1, \ldots$ such that for all $u > 0$, the state q_u is a successor of q_{u-1}. A computation $\lambda \in Q^\omega$ starting in state q is referred to as a *q-computation*; if $u \in \mathbb{N}$, then we denote by $\lambda[u]$ the u'th state in λ; similarly, we denote by $\lambda[0, u]$ and $\lambda[u, \infty]$ the finite prefix $q_0, \ldots, q_u$ and the infinite suffix $q_u, q_{u+1}, \ldots$ of λ respectively.

Intuitively, a *strategy* is an abstract model of an agent's decision-making process; a strategy may be thought of as a kind of plan for an agent. Formally, a strategy f_a for an agent $a \in \Sigma$ is a total

function $f_a : Q^+ \rightarrow 2^Q$, which must satisfy the constraint that $f_a(\lambda \cdot q) \in \delta(q, a)$ for all $\lambda \in Q^*$ and $q \in Q$. Given a set $C \subseteq \Sigma$ of agents, and an indexed set of strategies $F_C = \{f_a \mid a \in C\}$, one for each agent $a \in C$, we define $out(q, F_C)$ to be the set of possible outcomes that may occur if every agent $a \in C$ follows the corresponding strategy f_a, starting when the system is in state $q \in Q$. That is, the set $out(q, F_C)$ will contain all possible q-computations that the agents C can "enforce" by cooperating and following the strategies in F_C. Note that the "grand coalition" of all agents in the system can cooperate to uniquely determine the future state of the system, and so $out(q, F_\Sigma)$ is a singleton. Similarly, the set $out(q, F_\emptyset)$ is the set of all possible q-computations of the system.

We can now give the rules defining the satisfaction relation "$\models$" for ATL, which holds between pairs of the form S, q (where S is an ATS and q is a state in S), and formulae of ATL:

$S, q \models \top$;

$S, q \models p$ iff $p \in \pi(q)$ (where $p \in \Pi$);

$S, q \models \neg\varphi$ iff $S, q \not\models \varphi$;

$S, q \models \varphi \vee \psi$ iff $S, q \models \varphi$ or $S, q \models \psi$;

$S, q \models \langle\langle C \rangle\rangle \bigcirc \varphi$ iff there exists a set of strategies F_C, such that for all $\lambda \in out(q, F_C)$, we have $S, \lambda[1] \models \varphi$;

$S, q \models \langle\langle C \rangle\rangle \Box \varphi$ iff there exists a set of strategies F_C, such that for all $\lambda \in out(q, F_C)$, we have $S, \lambda[u] \models \varphi$ for all $u \in \mathbb{N}$;

$S, q \models \langle\langle C \rangle\rangle \varphi \,\mathcal{U}\, \psi$ iff there exists a set of strategies F_C, such that for all $\lambda \in out(q, F_C)$, there exists some $u \in \mathbb{N}$ such that $S, \lambda[u] \models \psi$, and for all $0 \leq v < u$, we have $S, \lambda[v] \models \varphi$.

Satisfaction for the ATL formula $\langle\langle C \rangle\rangle \Diamond \varphi$ is defined via equivalence to $\langle\langle C \rangle\rangle \top \,\mathcal{U}\, \varphi$.

We identify two important fragments of ATL which we will subsequently refer to:

- *Propositional logic* is the fragment of ATL in which no cooperation modalities are permitted. The truth of a propositional logic formula is determined on an ATS by using the first four items of the definition for satisfaction above.
- *Coalition Logic* is the fragment of ATL in which the only cooperation modalities allowed are of the form $\langle\langle C \rangle\rangle \bigcirc$ [19, 20, 9]. The truth of a Colition Logic formula is determined on an ATS by using the first five items of the definition for satisfaction above.

The (explicit state) *model checking* problem[1] is the problem of determining, for a given formula φ and ATS $S = \langle \Pi, \Sigma, Q, Q_0, \pi, \delta \rangle$, whether $S, q \models \varphi$ for all $q \in Q_0$. The model checking problems for ATL (and hence Coalition Logic) can be solved by a deterministic algorithm in time polynomial in the size of the inputs ATS S and the formula φ [2, pp.690–691] – assuming that ATS S in question is explicitly enumerated in the input in all its components, and the number of agents is considered fixed [13].

For the *satisfiability* problem, we are simply given a formula φ and asked whether there is an ATS S and state q in S such that $S, q \models \varphi$. The satisfiability problem for ATL is EXPTIME-complete [7, 25], while for Coalition Logic it is PSPACE-complete in the general case [19, p.63].

3. SRML

We have already observed that the positive tractability results for ATL and Coalition Logic model checking rely on the assumption that the model against which a formula is to be checked is explicitly enumerated in the input to the model checking algorithm, and that this assumption is both infeasible (there will be exponentially many states in the number of Boolean variables), and never used in practice. Instead, practical model checkers such as SMV [17], SPIN [12], and MOCHA [3] use model specification languages which permit the *succinct*, *high-level* specification of models. In this section, we describe the language used in this paper for this purpose: the Simple Reactive Modules Language (SRML). As its name suggests, this language is a strict subset of the RML language used in the MOCHA model checker[2]. Note that our choice of language is not arbitrary: we comment on it below. We begin with an informal overview.

3.1 Overview

Agents in RML are known as *modules*, and for consistency with the pre-existing literature on RML, we will stay with this terminology in SRML. An SRML module consists of:

- an *interface*, which defines the name of the module and lists the Boolean variables under the *control* of the module; and
- a number of *guarded commands*, which define the choices available to the module at every state.

The states of an SRML-defined model correspond to the possible valuations that may be given to the Boolean variables under the control of modules in the system: for the purposes of SRML, two states that agree on the valuation of all variables are the same[3].

The guarded commands of a module are divided into two sets: those used for *initialising* the variables under the module's control (`init` guarded commands), and those used for *updating* these variables subsequently (`update` guarded commands). A guarded command has two parts: a "condition" part (the "guard"), and a corresponding "action" part, which defines how to update the value of (some of) the variables under the control of a module. The intuitive reading of a guarded command $\varphi \rightsquigarrow e$ is "if the condition φ is satisfied, then *one* of the choices available to the module is to execute the assignment expression e". The assignment expression e is essentially a sequence of assignment statements, just as in conventional imperative programming languages: these assignment statements define how (some subset of) the module's controlled variables should be updated if the guarded command is executed. Notice that the truth of the guard φ does not mean that e *will* be executed: only that it is *enabled* for execution – it *may* be executed. If no guarded commands of a given module are enabled in some state, then the values of the variables in that module are assumed to remain unchanged in the next state: the module has no choice.

[1] Model checking is sometimes formulated as the problem of finding the set of states in a structure that satisfy a given formula; our formulation is chosen to be consistent with the way in which actual model checkers work, and it should be stressed that from the asymptotic analysis point of view, there is no difference in complexity between the formulations.

[2] For those familiar with RML and MOCHA, SRML assumes finite, propositional modules, with each module containing a single atom, in which no variable is awaited by any module.

[3] This is of course not the case in general in modal/temporal logic.

The choices available to a module wrt initialisation of its variables are thus defined by the `init` guarded commands of the module, while the choices available to a module wrt updating its variables are defined by the `update` guarded commands. To make this more concrete, here is an example guarded command:

$$\underbrace{(x \wedge y)}_{\text{guard}} \rightsquigarrow \underbrace{x' := \bot; y' := \top}_{\text{action}}$$

The guard is the propositional logic formula $x \wedge y$, so this guarded command will be enabled in any system state where both x and y take the value "$\top$". If the guarded command is chosen, then the effect is that in the next state of the system, the variable x will take value $\bot$, while y will take value $\top$. (The "prime" notation v' means "the value of variable v *after* the statement is executed".) We will write `skip` as an abbreviation for the empty assignment expression, which leaves the value of every variable controlled by a module unchanged.

Here is an example of a module, illustrating the concrete syntax that we will use for modules (which is essentially that of RML [1]).

```
module toggle controls x
  init
  [] ⊤ ⇝ x' := ⊤
  [] ⊤ ⇝ x' := ⊥
  update
  [] x ⇝ x' := ⊥
  [] (¬x) ⇝ x' := ⊤
```

This module, named *toggle*, controls a single Boolean variable, x. There are two `init` guarded commands and two `update` guarded commands. (The symbol "`[]`" is a syntactic separator.) The `init` guarded commands of *toggle* define two choices for the initialisation of this variable: assign it the value $\top$ or the value $\bot$. With respect to `update` guarded commands, the first command says that if x has the value $\top$, then the corresponding choice is to assign it the value $\bot$, while the second command says that if x has the value $\bot$, then it can subsequently be assigned the value $\top$. In other words, the module non-deterministically chooses a value for x initially, and then on subsequent rounds toggles this value. Notice that in this example, the `init` commands of this module are non-deterministic, while the `update` commands are deterministic.

3.2 Formal Definition

In this section, we will use Π to denote the set of Boolean variables in a system, as in ATL. Formally, a guarded command γ over Π is an expression

$$\gamma : \qquad \varphi \rightsquigarrow v_1' := \psi_1; \ldots; v_k' := \psi_k$$

where φ (the guard) is a propositional logic formula over Π, each v_i is a member of Π and ψ_i is a propositional logic formula over Π. We require that no variable v_i appears on the l.h.s. of two assignment statements in the same guarded command (hence no issue on the ordering of the updates arises). The intended interpretation is that if the formula φ evaluates to true against the interpretation corresponding to the current state of the system, then the statement is *enabled* for execution; executing the statement means evaluating each ψ_i against the current state of the system, and setting the corresponding variable v_i to the truth value obtained from evaluating ψ_i. We say that $v_1, \ldots, v_k$ are the *controlled variables* of γ, and denote this set by $ctr(\gamma)$. A set of guarded commands is said to be *disjoint* if their controlled variables are mutually disjoint.

Given a propositional valuation $\theta \subseteq \Pi$ and a guarded command $\gamma : \varphi \rightsquigarrow v_1' := \psi_1; \ldots; v_k' := \psi_k$ such that θ enables γ (i.e., $\theta \models \varphi$) we denote the result of *executing* γ on θ by $\theta \oplus \gamma$. For example, if $\theta = \{p, r\}$, and $\gamma = p \rightsquigarrow q' := \top; r' := p \wedge \neg r$, then $\theta \oplus \gamma = \{p, q\}$.

Given a propositional valuation $\theta \subseteq \Pi$, and set Γ of disjoint guarded commands over Π such that every member of Γ is enabled by θ, then the interpretation θ' resulting from Γ on θ is denoted by $\theta' = \theta \oplus \Gamma$: since the members of Γ are disjoint, we can pick them in any order to execute on θ.

As described above, there are two classes of guarded commands that may be declared in an atom: `init` and `update`. An `init` guarded command is only used to initialise the values of variables, when the system begins execution. Full RML allows for quite sophisticated initialisation schemes, but in SRML, we will assume that the guards to `init` command are "$\top$", i.e., every `init` command is enabled for execution in the initialisation round of the system.

An SRML *module*, m, is a triple:

$$m = \langle ctr, init, update \rangle \text{ where:}$$

- $ctr \subseteq \Pi$ is the (finite) set of variables controlled by m;
- $init$ is a (finite) set of *initialisation* guarded commands, such that for all $\gamma \in init$, we have $ctr(\gamma) \subseteq ctr$; and
- $update$ is a (finite) set of *update* guarded commands, such that for all $\gamma \in init$, we have $ctr(\gamma) \subseteq ctr$.

Given a module m, we denote the controlled variables of m by $ctr(m)$, the initialisation guarded commands of m by $init(m)$, and the update guarded commands of m by $update(m)$. An SRML system is then an $(n+2)$-tuple

$$\langle \Sigma, \Pi, m_1, \ldots, m_n \rangle$$

where $\Sigma = \{1, \ldots, n\}$ is a set of agents and Π is a vocabulary of Boolean variables, (as in ATSs), and for each $i \in \Sigma$, m_i is the corresponding module defining i's choices; we require that $\{ctr(m_1), \ldots, ctr(m_n)\}$ forms a partition of Π (i.e., every variable in Π is controlled by some agent, and no variable is controlled by more than one agent).

A *joint* guarded command (j.g.c.) for a coalition $C \subseteq \Sigma$ is an indexed tuple $\langle \gamma_1, \ldots, \gamma_k \rangle$, with a guarded command $\gamma_i \in m_i$ for each $i \in C$. A j.g.c. $\langle \gamma_1, \ldots, \gamma_k \rangle$ is enabled by a propositional valuation $\theta \subseteq \Pi$ iff all its members are enabled by θ.

Given an SRML system R, the corresponding ATS, which we denote by S_R, is defined in the obvious way:

- the states Q of S_R correspond to the possible valuations to variables Π, with Q_0 being the subset of Q that may be generated by the $init$ guarded commands of the modules in R; and
- the δ function of S_R is defined in each state for each possible coalition by the sets of enabled guarded commands (choices) for that coalition.

We write $R \models \varphi$ to indicate that $S_R, q_0 \models \varphi$ for every initial state q_0 in S_R: the SRML model checking problem is then the problem of checking, for given R and φ, whether or not $R \models \varphi$.

4. MAIN RESULTS

We now present our main results. Given the concerns of this paper – understanding the complexity of ATL model checking for a particular model representation – the first of these results may seem somewhat strange, but its role will become clear shortly. We show that, given an SRML system R, we can construct a formula

$mimic(R)$ of ATL such that $mimic(R)$ is *only* satisfied in models that are "equivalent" to S_R. We can thus understand $mimic(R)$ as representing the ATL *theory* of R. An important property of the construction of $mimic(R)$ is that this formula is of size polynomial in the size of R. For such a $mimic(R)$, we must characterize the behaviour of R in the sense that:

(a) abilities in R are transferred to models satisfying $mimic(R)$;

(b) agents should not be able to achieve more in any $mimic(R)$-model than the modules can do in R.

The construction of $mimic(R)$ is as follows. Let the variables $ctr(i)$ under control of module i be $v_1^i, \ldots, v_j^i$, and let the module i be $\langle ctr(i), init, update\rangle$. We are now going to define requirements $t(\mathtt{init}, i)$, (2), (3) and (4); they are supposed to take care of our condition (a). Regarding $init$, we do the following. Suppose there are c_i *init* guarded commands for agent i, and let a typical $init$ guarded command ι_a^i ($a \leq c_i$) for agent i be composed of a positive part $\iota_{a_u}^{+_i}$ and a negative part $\iota_{a_v}^{-_i}$ ($u \leq nu, v \leq nv$) as follows:

$$\iota_{a_u}^{+_i} \quad \top \rightsquigarrow v_{a_1}^{+_i} := \top, \ldots, \top \rightsquigarrow v_{a_{nu}}^{+^i} := \top,$$

$$\iota_{a_v}^{-_i} \quad \top \rightsquigarrow v_{a_1}^{-_i} := \bot, \ldots, \top \rightsquigarrow v_{a_{nv}}^{-_i} := \bot$$

Where no $v_{a_j}^{+_i}$ can be the same as any $v_{a_k}^{+_i}$, for any $j \leq nu$ and $k \leq nv$ (this is guaranteed by the fact that $\iota_a^{+_i}$ and $\iota_a^{-_i}$ represent one guarded command). Then denote

$$t(\mathtt{init}, i) = \bigvee_{a \leq c_i} (\bigwedge_{j \leq nu} v_{a_j}^{+_i} \wedge \bigwedge_{k \leq nv} \neg v_{a_k}^{-_i}) \tag{1}$$

This formula $t(\mathtt{init}, i)$ expresses the effect of executing one of the c_i guarded commands ι_a for agent i. Note that we do not have an *exclusive* disjunction here, since it might well be that the *effect* of executing one initialsation command ι_a may coincide with that of another. Given an SRML-system R, it is obvious that in its associated ATS S_R, a state q verifies $t(\mathtt{init}, i)$ iff q is one of the initial states in Q_0.

Now we turn to the $update$ part of the module. Suppose it looks like the following (note that all $v_j'^i$ are occurrences of i's variables, not the variables themselves).

$$\begin{array}{lllll}
\gamma_1^i: & \varphi_1 \rightsquigarrow & {v_{1_1}^i}' := \psi_{1_1} & \ldots & {v_{1_{n1}}^i}' := \psi_{1_{n1}} \\
\gamma_2^i: & \varphi_2 \rightsquigarrow & {v_{2_1}^i}' := \psi_{2_1} & \ldots & {v_{2_{n2}}^i}' := \psi_{2_{n2}} \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
\gamma_m^i: & \varphi_m \rightsquigarrow & {v_{m_1}^i}' := \psi_{m_1} & \ldots & {v_{m_{nm}}^i}' := \psi_{m_{nm}}
\end{array}$$

Given an SMRL R, introduce for every ψ_{k_t} ($k \leq m, t \leq nk$) a new propositional variable x_{k_t}. The idea is that every x will store the old value of its corresponding ψ, so we stipulate:

$$\bigwedge_{k \leq m, t \leq nk} \langle\langle\rangle\rangle \Box (\psi_{k_t} \leftrightarrow \langle\langle\rangle\rangle \bigcirc x_{k_t}) \tag{2}$$

Some of i's variables will not be changed by executing a command, so we also need to keep track of the variables old values. For that, we introduce, for every $v_e^i \in ctr(i)$, variables y_e^i ($e \leq j$), and stipulate

$$\bigwedge_{e \leq j} \langle\langle\rangle\rangle \Box (v_e^i \leftrightarrow \langle\langle\rangle\rangle \bigcirc y_e^i) \tag{3}$$

We now look at our constraint (b), which should take care of the fact that the if σ is ATL-satisfiable, it should be in a context that is behaving well enough in order to be transferred to an SMRL R. To guarantee that an agent i cannot spontaneously bring about anything that is not triggered by the execution of a command γ_z^i, we introduce propositional variables $done(i,1), \ldots, done(i,m)$ and a special atom $none(i)$. The latter atom denotes that i did not execute any of its guarded commands. Now consider a command γ_z^i. As before, let $ctr(\gamma_z^i)$ be i's variables occurring in the command γ_z^i. Now represent γ_z^i by

$$\varphi_z \rightarrow \langle\langle i\rangle\rangle \bigcirc (new(i) \wedge rec(i,z)) \tag{4}$$

where

$$new(i) = \bigwedge_{v \in ctr(\gamma_z^i)} (v \leftrightarrow x) \wedge \bigwedge_{u \in ctr(i) \setminus ctr} (u \leftrightarrow y)$$

and

$$rec(i,z) = done(i,z) \wedge \bigwedge_{s \leq m, s \neq z} \neg done(i,s) \wedge \neg none(i)$$

$new(i)$ says that the values of the relevant ψ_{k_t} are, via x_{k_t}, copied in the corresponding variable $v_{k_t}^i$, and the other variables u of i receive their 'old' values through a copy of the corresponding variables y. This takes care of our goal specified as (a). For the (b)-part, $rec(i,z)$ takes a record of which guarded command has been applied. This ensures that agent i indeed 'only' executes guarded commands, if we add the global properties (5)–(8):

$$\langle\langle\rangle\rangle \Box ((\bigwedge_{k \leq m} \neg \varphi_k) \leftrightarrow \langle\langle\rangle\rangle \bigcirc none(i)) \tag{5}$$

$$\langle\langle\rangle\rangle \Box \bigwedge_{k \leq m} (\langle\langle\rangle\rangle \bigcirc done(i,k) \rightarrow \varphi_k) \tag{6}$$

$$\langle\langle\rangle\rangle \Box \bigwedge_{k \leq m} (done(i,k) \rightarrow (new(i) \wedge \neg none(i))) \tag{7}$$

$$\begin{array}{ll}
\langle\langle\rangle\rangle \Box (& (\neg none(i) \rightarrow \bigtriangledown_{k \leq m} done(i,k)) \\
& \wedge (none(i) \rightarrow \bigwedge_{v \in ctr(i)} v \leftrightarrow y) \ )
\end{array} \tag{8}$$

Equation (5) expresses that agent i will only do nothing if all his guards are false; (6) says that i can only execute the k-th command if the k-th guard is true; (7) expresses that if a flag is set indicating that i executes the k-th command, then, indeed, this command is executed, and the $none(i)$-flag is not set to true, and, finally, (8) expresses that either i does nothing, or he executes exactly one of his guarded commands and that doing nothing implies keeping all the "old" values for i's variables.

We are now in a position to define $mimic(R)$:

$$\bigwedge_{i \leq n} (t(\mathtt{init}, i) \wedge (2) \wedge (3) \wedge (4) \wedge (5) \wedge (6) \wedge (7) \wedge (8))$$

Note that the size of $mimic(R)$ is polynomial in R, since we only add a given number of new variables, where the number depends on the number of local variables and the number of formulas ψ appearing at the right of a ":=" in any command, and each of the conjuncts in $mimic(R)$ is polynomial in R.

Now, suppose we have some S, q such that $S, q \models mimic(R)$. The states of S will not look exactly like the states of S_R, since

they will include propositional variables that will not be in S_R (i.e., the new variables we introduced in the translation). But nevertheless, the properties of agents – their abilities, with respect to the propositional variables of R – must be the same in the ATS S that satisfies $mimic(R)$ as in R. In short: *the model S must preserve the truth of* ATL *formulae over R.*

Returning to the main concerns of the paper, we now see the relevance of this construction.

THEOREM 1. *The* ATL *model checking problem for* SRML *is polynomial-time reducible to* ATL *unsatisfiability.*

PROOF. Let R be the given SRML system, and φ be the formula we want to check. Then the following are equivalent:

1. $R \models \varphi$
2. $mimic(R) \rightarrow \varphi$ is valid in ATL.

□

The rationale for stating this result will become clear in the following:

THEOREM 2. *The* ATL *model checking problem for* SRML *models is* EXPTIME*-complete.*

PROOF. Membership of EXPTIME follows from the fact, proved in Theorem 1, that the ATL model checking problem for SRML is polynomial time reducible to ATL unsatisfiability. Since the ATL satisfiability problem is EXPTIME complete [7, 25], then ATL unsatisfiability is in co-EXPTIME; but since EXPTIME is a deterministic complexity class, EXPTIME = co-EXPTIME, and we conclude that the SRML model checking for SRML is in EXPTIME.

We prove EXPTIME-hardness by reduction from the problem of determining whether a given player has a winning strategy in the two-player game PEEK-G_4 [24, p.158]. An instance of PEEK-G_4 is a quad:

$$\langle X_1, X_2, X_3, \varphi \rangle \text{ where:}$$

- X_1 and X_2 are disjoint, finite sets of Boolean variables, with the intended interpretation that the variables in X_1 are under the control of agent 1, and X_2 are under the control of agent 2;
- $X_3 \subseteq (X_1 \cup X_2)$ are the variables deemed to be true in the initial state of the game; and
- φ is a propositional logic formula over the variables $X_1 \cup X_2$, representing the winning condition.

The game is played in a series of rounds, with the agents $i \in \{1, 2\}$ alternating (with agent 1 moving first) to select a value (true or false) for one of their variables in X_i, with the game starting from the initial assignment of truth values defined by X_3. Variables that were not changed retain the same truth value in the subsequent round. An agent wins in a given round if it makes a move such that the resulting truth assignment defined by that round makes the winning formula φ true. The decision problem associated with PEEK-G_4 involves determining whether agent 2 has a winning strategy in a given game instance $\langle X_1, X_2, X_3, \varphi \rangle$. Notice that PEEK-$G_4$ only requires "memoryless" (Markovian) strategies: whether or not an agent i can win depends only on the current truth assignment, the distribution of variables, the winning formula, and whose turn it is currently. As a corollary, if agent i can force a win, then it can force a win in $O(2^{|X_1 \cup X_2|})$ moves.

Given an instance $\langle X_1, X_2, X_3, \varphi \rangle$ of PEEK-G_4, we produce an instance of SRML model checking as follows. For each Boolean variable $x \in (X_1 \cup X_2)$, we create a variable with the same name in our SRML model, and we create an additional Boolean variable $turn$, with the intended interpretation that if $turn = \top$, then it is agent 1's turn to move, while if $turn = \bot$, then it is agent 2's turn to move. We have a module $move$, the purpose of which is to control $turn$, toggling its value in each successive round, starting from the initial case of it being agent 1's move.

$$\begin{array}{l}
\texttt{module}\ move\ \texttt{controls}\ turn \\
\quad \texttt{init} \\
\quad [\!]\top \rightsquigarrow turn' := \top \\
\quad \texttt{update} \\
\quad [\!]\, turn \rightsquigarrow turn' := \bot \\
\quad [\!](\neg turn) \rightsquigarrow turn' := \top
\end{array}$$

For each of the two PEEK-G_4 players $i \in \{1, 2\}$, we create an SRML module ag_i that `controls` the variables X_i. The module ag_i is as follows. It begins by deterministically initialising the values of all its variables to the values defined by X_3 (that is, if variable $x \in X_i$ appears in X_3 then this variable is initialised to $\top$, otherwise it is initialised to $\bot$). Subsequently, when it is this player's turn, it can non-deterministically choose at most one of the variables under its control and toggle the value of this variable; when it is not this player's turn, it has no choice but to do nothing, leaving the value of all its variables unchanged. The general structure of ag_1 is thus as follows, where $X_1 = \{x_1, \ldots, x_k\}$.

$$\begin{array}{l}
\texttt{module}\ ag_1\ \texttt{controls}\ x_1, \ldots, x_k \\
\quad \texttt{init} \\
\quad \texttt{// initialise to values from}\ X_3 \\
\quad [\!]\top \rightsquigarrow x_1' := \ldots; x_k := \ldots \\
\quad \texttt{update} \\
\quad [\!]\, turn \rightsquigarrow x_1' := \bot \\
\quad [\!]\, turn \rightsquigarrow x_1' := \top \\
\quad \ldots \\
\quad [\!]\, turn \rightsquigarrow x_k' := \bot \\
\quad [\!]\, turn \rightsquigarrow x_k' := \top \\
\quad [\!]\top \rightsquigarrow \texttt{skip}
\end{array}$$

Notice that an agent can always `skip`, electing to leave its variables unchanged; and, if it is not this agent's turn to move, this is the *only* choice it has.

The SRML system under consideration contains just these three modules. It can be shown easily that player 2 has a strategy for φ in the PEEK-G_4 game $\langle X_1, X_2, X_3, \varphi \rangle$ iff the SRML system satisfies the formula $\langle\langle 2 \rangle\rangle (\neg\varphi)\, \mathcal{U}\, (\varphi \wedge turn)$. □

There are several small points of interest about this proof. First, notice that the reduction used for EXPTIME-hardness requires *only a fixed number of agents* (three). Thus we have the following, stronger result: ATL model checking for SRML models is EXPTIME complete for any fixed number n of modules, where $n \geq 3$. Second, notice the form of the guarded commands used in the reduction: on the rhs of any guard, we only change the value of at most one variable. Thus, again, we have a stronger result: EXPTIME hardness even when guarded commands are assumed to be of this particularly impoverished form.

Next, we turn our attention to significant subsets of ATL: first, Coalition Logic, the fragment in which the only temporal modality allowed is "$\bigcirc$" [19, 21, 9].

THEOREM 3. *The Coalition Logic model checking problem for* SRML *models is* PSPACE*-complete.*

PROOF. We first prove PSPACE-hardness, by reducing the problem of determining the truth of Quantified Boolean Formulae (QBF) [18, pp.456–458] to that of Coalition Logic model checking against SRML models.

An instance of QBF is given by a formula

$$\exists x_1 \forall x_2 \cdots Q_k \varphi(x_1, x_2, \ldots, x_k)$$

where $x_1, \ldots, x_k$ are Boolean variables, the quantifier Q_k is $\exists$ if k is odd, and $\forall$ if k is even, and $\varphi(x_1, \ldots, x_k)$ is a propositional logic formula over the variables $x_1, x_2, \ldots, x_k$. Such a formula is true if there exists a valuation for x_1 such that for all valuations for x_2, ..., such that the formula $\varphi(x_1, x_2, \ldots, x_k)$ is true. We proceed to create an SRML system containing three modules, $move$, $ag_\exists$, $ag_\forall$, the variables $x_1, x_2, \ldots, x_k$, and, in addition, k variables $m_1, m_2, \ldots, m_k$, such that m_i will be true if we are about to assign a value for variable x_i.

The module $turn$ simply passes the value $\top$ along each of the variables $m_1, m_2, \ldots, m_k$ in turn.

```
module turn controls m_1, ..., m_k
  init
  [] ⊤ ⇝ m_1' := ⊤, m_2' := ⊥; ...; m_k := ⊥
  update
  [] m_1 ⇝ m_1' := ⊥; m_2' := ⊤
  ...
  [] m_i ⇝ m_i' := ⊥; m_{i+1}' := ⊤
  ...
  [] m_{k-1} ⇝ m_{k-1}' := ⊥; m_k' := ⊤
```

Notice that at most one of the variables $m_1, m_2, \ldots, m_k$ will be true at any given time.

We define the modules $ag_\exists$ as follows. This module `controls` all odd numbered variables. On even rounds, the module simply `skip`s, leaving the values of its variables unchanged. On odd numbered rounds i, $1 \leq i \leq k$, the module will have two choices: make variable x_i true or make x_i false.

```
module ag_∃ controls ...// odd numbered variables
  init
  [] ⊤ ⇝ skip
update
  [] m_1 ⇝ x_1' := ⊥
  [] m_1 ⇝ x_1' := ⊤
  [] m_2 ⇝ skip
  [] m_3 ⇝ x_3' := ⊥
  [] m_3 ⇝ x_3' := ⊤
  [] m_4 ⇝ skip
  ...
```

We define $ag_\forall$ similarly, swapping "even" for "odd". Noting that (i) the module $turn$ plays no part in determining the value of variables other than the move variables $m_1, \ldots, m_k$; (ii) when it is player i's turn in round j, then the only choices i has are to assign truth or falsity to variable x_j; (iii) when it is not player is turn in round j, then j must leave all its variables unchanged; (iv) that the construction is clearly polynomial in the size of the input formula.

The QBF instance $\exists x_1 \forall x_2 \cdots Q_k \varphi(x_1, x_2, \ldots, x_k)$ is true iff the the SRML system above satisfies the formula

$$\langle\langle ag_\exists \rangle\rangle \bigcirc \neg \langle\langle ag_\forall \rangle\rangle \bigcirc \neg \cdots \varphi(x_1, x_2, \ldots, x_k).$$

To see that the model checking problem is decidable in PSPACE, we present an algorithm for deciding the problem that works in polynomial space: see Figure 1. The algorithm takes as input an

```
1.  function eval(φ, R) returns ⊤ or ⊥
2.    θ := ∅
3.    for each θ-enabled initial j.g.c. ⟨γ_1, ..., γ_n⟩ for Σ do
4.      if not aux(φ, θ ⊕ {γ_1, ..., γ_n}, R) then
5.        return ⊥
6.      end-if
7.    end-for
8.    return ⊤
9.  end-function eval

10. function aux(φ, θ, R) returns ⊤ or ⊥
11.   if φ ∈ Π then
12.     if φ ∈ θ then return ⊤ else return ⊥ end-if
13.   elsif φ = ¬ψ then
14.     return not aux(ψ, θ, R)
15.   elsif φ = ψ_1 ∨ ψ_2 then
16.     return aux(ψ_1, θ, R)
17.       or aux(ψ_2, θ, R)
18.   elsif φ = ⟨⟨C⟩⟩◯ψ then
19.     for each θ-enabled update j.g.c. ⟨γ_1, ..., γ_l⟩ for C do
20.       flag := ⊤
21.       for each θ-enabled update j.g.c. ⟨γ_{l+1}, ..., γ_n⟩ for Σ \ C do
22.         if not aux(ψ, θ ⊕ {γ_1, ..., γ_l, γ_{l+1}, ..., γ_n}, R) then
23.           flag := ⊥
24.         end-if
25.       end-for
26.       if flag then return ⊤ end-if
27.     end-for
28.     return ⊥
29.   end-if
30. end-function aux
```

Figure 1: A polynomial space algorithm for checking Coalition Logic formulae against SRML models.

SRML system R, and a formula φ of coalition logic to check against this system. The algorithm is in two parts: the first part (the function $eval(\cdots)$) generates each initial state of the system in turn from the initial guarded commands of the modules comprising R, and then invokes the auxiliary function $aux(\cdots)$ to check whether the input φ is true in these initial states. The $aux(\cdots)$ function is recursive; we note that the number of recursive calls will be bounded by the size of the input formula φ, with each call requiring only polynomial space. Termination and correctness are immediate from construction; it only remains to note that we can loop through the enabled initial guarded commands (lines 3–7) and the enabled update guarded commands (lines 19–23) in polynomial space. □

So, under standard complexity theoretic assumptions, model checking Coalition Logic against SRML models is "easier" than full ATL. Suppose we consider other restrictions on the logic that we check.

THEOREM 4. *The propositional logic model checking problem for* SRML *is co-*NP*-complete.*

PROOF. For membership, simply note that we can universally select all initial j.g.c.s $\langle \gamma_1, \ldots, \gamma_m \rangle$, and check that if $\langle \gamma_1, \ldots, \gamma_m \rangle$ is satisfied by the empty truth assignment $\theta_\emptyset$, then the interpretation $\theta_\emptyset \oplus \{\gamma_1, \ldots, \gamma_m\}$ satisfies φ. For completeness, we can reduce the problem TAUT, of checking that a propositional logic formula is a tautology, i.e., satisfied under all truth assignments. For each Boolean variable x appearing in the input instance φ, we create an agent controlling x, with two initial guarded commands, both enabled by the empty assignment $\theta_\emptyset$, which set x to $\top$ and $\bot$ respectively. The formula to be checked against this system is simply the TAUT instance φ. The initial states of the system thus constructed

correspond to all valuations of the variables of φ, and hence φ is a tautology iff φ holds in all initial states of the system. □

5. DISCUSSION

In this paper, we have shown that, for three important subsets of ATL (including ATL itself), the model checking problem has exactly the same complexity as the corresponding theorem proving problem, assuming that models are represented using SRML, a simplified version of RML [1]. This seems to us to be a striking result: *practical* model checking for ATL and Coalition Logic has the same complexity as theorem proving for these logics. We note that others have pointed out that ATL model checking is more complex than it might appear [13], and other results hint that practical ATL model checking might be more complex than it appears at first sight [5, 6]: we have obtained tight bounds on the complexity of these problems for a specific, practical representation for models.

It is important to note that the SRML language is *not* contrived: it is a strict subset of the RML language that is used by several practical model checking systems, including MOCHA, the ATL model checker [3]. Indeed, SRML is arguably the smallest "useful" subset of RML that one can imagine: it is hard to imagine how one could simplify it without making it unusable in practice. It is also worth noting that the guarded command structures used to define SRML models are also used (modulo syntactic differences) for the same purpose in other model checking systems such as SPIN [12] and SMV [17].

We note that one of the motivations for developing the Coalition Logic of Propositional Control (CL-PC), a simpler modal variation of ATL [11], was that the structure of controlled variables in languages like RML permitted a more direct (and simpler) semantics to cooperation modalities. This link – between the interpretation of cooperation modalities and the languages used to define models – is worth exploring further.

6. REFERENCES

[1] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(11):7–48, July 1999.

[2] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, September 2002.

[3] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Taşiran. Mocha: Modularity in model checking. In *CAV 1998: Tenth International Conference on Computer-aided Verification, (LNCS Volume 1427)*, pages 521–525. Springer-Verlag: Berlin, Germany, 1998.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.

[5] L. de Alfaro, T. A. Henzinger, , and F. Y. C. Mang. The control of synchronous systems. In *CONCUR 2000: Concurrency Theory, 11th International Conference (LNCS Volume 1877)*, pages 458–473. Springer-Verlag: Berlin, Germany, 2000.

[6] L. de Alfaro, T. A. Henzinger, , and F. Y. C. Mang. The control of synchronous systems part ii. In *CONCUR 2001: Concurrency Theory, 12th International Conference (LNCS Volume 2154)*, pages 566–580. Springer-Verlag: Berlin, Germany, 2001.

[7] G. van Drimmelen. Satisfiability in alternating-time temporal logic. In *Eighteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2003)*, pages 208–217, Ottawa, Canada, 2003.

[8] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[9] V. Goranko. Coalition games and alternating temporal logics. In J. van Benthem, editor, *Proceeding of the Eighth Conference on Theoretical Aspects of Rationality and Knowledge (TARK VIII)*, pages 259–272, Siena, Italy, 2001.

[10] J. Y. Halpern and M. Y. Vardi. Model checking versus theorem proving: A manifesto. In V. Lifschitz, editor, *AI and Mathematical Theory of Computation — Papers in Honor of John McCarthy*, pages 151–176. The Academic Press: London, England, 1991.

[11] W. van der Hoek and M. Wooldridge. On the logic of cooperation and propositional control. *Artificial Intelligence*, 164(1-2):81–119, May 2005.

[12] G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[13] W. Jamroga and J. Dix. Do agents make model checking explode (computationally)? In M. Pechoucek, P. Petta, and L. Z. Varga, editors, *Multi-Agent Systems and Applications IV (LNAI Volume 3690)*, 2005.

[14] W. Jamroga and W. van der Hoek. Agents that know how to play. *Fundamenta Informaticae*, 63(2-3):185–219, 2004.

[15] M. Kacprzak and W. Penczek. A SAT-based approach to unbounded model checking for alternating-time temporal epistemic logic. *Synthese*, 142(2):203–227, November 2004.

[16] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[17] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers: Dordrecht, The Netherlands, 1993.

[18] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley: Reading, MA, 1994.

[19] M. Pauly. *Logic for Social Software*. PhD thesis, University of Amsterdam, 2001. ILLC Dissertation Series 2001-10.

[20] M. Pauly. A logical framework for coalitional effectivity in dynamic procedures. *Bulletin of Economic Research*, 53(4):305–324, 2002.

[21] M. Pauly. A modal logic for coalitional power in games. *Journal of Logic and Computation*, 12(1):149–166, 2002.

[22] F. Raimondi and A. Lomuscio. The complexity of symbolic model checking temporal-epistemic logics. In *Proceedings of Concurrency, Specification & Programming (CS&P)*, Ruciane-Nida, Poland, September 2005.

[23] M. Ryan and P.-Y. Schobbens. Agents and roles: Refinement in alternating-time temporal logic. In J.-J. Ch. Meyer and M. Tambe, editors, *Intelligent Agents VIII: Proceedings of the Eigth International Workshop on Agent Theories, Architectures, and Languages, ATAL-2001 (LNAI Volume 2333)*, pages 100–114, 2002.

[24] L. J. Stockmeyer and A. K. Chandra. Provably difficult combinatorial games. *SIAM Journal of Computing*, 8(2):151–174, 1979.

[25] D. Walther, C. Lutz, F. Wolter, and M. Wooldridge. ATL satisfiability is indeed EXPTIME-complete, 2005. accepted for publication.