Reasoning about Strategies of Multi-Agent Programs

Mehdi Dastani
Institute of Information and Computer Sciences
Utrecht University
Utrecht, the Netherlands
mehdi@cs.uu.nl

Wojciech Jamroga
Computer Science and Communication
University of Luxembourg,
and Department of Computer Science
Clausthal University of Technology
wojtek.jamroga@uni.lu

ABSTRACT

Verification of multi-agent programs is a key problem in agent research and development. This paper focuses on multi-agent programs that consist of a finite set of BDI-based agent programs executed concurrently. We choose alternating-time temporal logic (ATL) for expressing properties of such programs. However, the original ATL is based on a synchronous model of multi-agent computation while most (if not all) multi-agent programming frameworks use asynchronous semantics where activities of different agents are interleaved. Moreover, unlike in ATL, our agent programs do not have perfect information about the current global state of the system. They are not appropriate subjects for modal epistemic logic either (since they do not know the global model of the system). We begin by adapting the semantics of ATL to the situation at hand; then, we consider the verification problem in the new setting and present some preliminary results.

Categories and Subject Descriptors

I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent Systems*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Modal logic*

General Terms

Languages, Algorithms, Verification

Keywords

Agent-oriented programming, strategic logic, model checking

1. INTRODUCTION

Specification and verification of agent-based systems is a key problem in multi-agent research and development [9]. More recently, there has been a shift focusing on the verification of *multi-agent programs*, i.e., programs that are developed to implement multi-agent systems [10, 15, 7]. These works aim at verifying safety and liveness properties in terms of the internals of individual agent programs such as beliefs, goals, and plans [6, 16, 5, 1, 3]. Examples of such properties are realism, various kinds of commitment strategies, and communication abilities.

Cite as: Reasoning about Strategies of Multi-Agent Programs, Mehdi Dastani and Wojciech Jamroga, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. XXX-XXX.

This paper focuses on the verification of multi-agent programs, each consisting of a finite set of BDI-based individual agent programs operating in a shared environment and executed concurrently. We propose a framework that allows developers of multi-agent programs to check if (a subset of) individual agent programs can achieve specific states of their shared environment together. One may for example be interested in checking if a specific subset of agent programs, acting in a blocksworld environment, can cooperate to build a tower of blocks or to prevent an existing tower from collapsing.

In order to develop such a verification framework, a logic used for reasoning about multi-agent programs must be grounded in the computation of these programs. We begin by presenting (in Section 2) a simple multi-agent programming language that includes programming constructs employed by most BDI-based agent-oriented languages (e.g., [15, 7, 10]). Then, we discuss the choice of a formalism for specification and verification of properties for multi-agent programs. There are many logics that can be used for this purpose. Likewise, we can choose among several different semantics of program execution. We list some options and make (sometimes tentative) choices in Section 3. In essence, we choose alternating-time temporal logic with uniform strategies, interpreted over labeled interpreted systems that arise from the asynchronous semantics of program execution.

Sections 4 and 5 present the formal framework that implements our choices. That is, in Section 4 we present an operational semantics for the programming language that generates appropriate models of multi-agent programs, and in Section 5 we define a variant of alternating-time temporal logic (cf., e.g., [2, 17]) that seems suitable for reasoning about their properties. The logic can be used to reason about possible executions of multi-agent programs, and temporal patterns which can be enforced by agents and their groups. In particular, it facilitates specification and verification of agents' abilities wrt safety and/or liveness properties (with and without fairness assumptions). Finally, in Section 6, we discuss the verification problem for the setting: we establish basic complexity results, and present some ideas for optimization of models.

2. PROGRAMMING LANGUAGE

In this section we present the syntax of a simple BDI-based multi-agent programming language. The basic ingredients of this language appear in the existing agent programming languages [15, 7, 10] so that the proposed approach can be considered as generic and applicable to other BDI-based programming languages. In particular, the language allows for implementation of individual agents with beliefs, goals, actions, plans, and planning rules. For the purpose of this paper, we assume that a multi-agent program consists of a set of programs for individual agents and a shared environment in which all agents can perform actions. The environment is repre-

sented by a set of literals. In consequence, it suffices to present the syntax of the programming language for individual agent programs, which we do in the following subsections.

2.1 Beliefs and Goals

The *beliefs* of an agent represent its information about its environment of action, while its *goals* represent situations the agent wants to bring about (not necessarily all at once).[1] For simplicity, we represent an agent's beliefs as a set of literals (for example, `{-carry, free_a}` represents the agent's belief that it is not carrying a block and that block `a` is free) and goals as a set of conjunctions of literals (e.g., `{on_c_a and on_b_c, on_b_a}` represents the agent's goal to have blocks `c` on `a` and `b` on `c`, and a second goal to have block `b` on `a`). It is assumed that different goals of an agent may not be achievable in one state. Moreover, the beliefs and goals of an agent are related to each other: if an agent believes p, then it will not pursue p as a goal (since p has already been achieved). The initial beliefs and goals of an agent are specified by its program.

2.2 Basic Actions

Basic actions specify the capabilities an agent has to achieve its goals. In this paper, we consider basic actions of an agent as being specified in terms of both the agent's beliefs as well as its external environment. In particular, we assume that each basic action of an agent is specified in terms of a set of pairs, each of which consists of two pre-conditions and two post-conditions, cf. Remark 1. One of the pre-conditions is a condition on the agent's beliefs and the second is condition on its external environment. One post-condition determines the update of the agent's beliefs and the second determines the update of its external environment. The pre- and post-conditions are sets of literals. For example, consider the specification of basic action `put_b_c`, which consists of two pairs of pre- and post-conditions.

```
[-on_b_c]{carry} put_b_c {-carry}[ on_b_c]
[ on_b_c]{carry} put_b_c {-carry}[floor_b]
```

In this specification, the pre- and post-conditions on the agent's beliefs are placed within curly brackets `{}` and the pre- and post-conditions on the agent's environment are placed within square brackets `[]`. The belief pre-conditions of both pairs indicate that the action of putting block `b` on block `c` can be executed if the agent believes it is carrying a block. Their corresponding post-conditions indicate that after the action execution the agent believes not carrying a block anymore. The environment pre- and post-conditions of the first pair indicate that the execution of the action puts block `b` on block `c` if it was not already the case. The conditions in the second pair indicate that if block `b` was already on block `c`, then the action `put_b_c` will cause block `b` to fall on the floor (presumably the agent believes wrongly that it is carrying block `b`, and handles it in a wrong way).

In general, a basic action can be executed only if one of its belief pre-conditions is derivable from the agent's beliefs. In such a case, we call the basic action *executable*. If none of the belief pre-conditions is derivable from the agent's beliefs, then the execution of the action blocks. The execution of an executable action will update the agent's beliefs with the corresponding belief post-condition of the action. We assume that basic actions maintain consistency of the agent's beliefs, and that different belief pre-conditions of one basic action cannot be satisfied in one state. The environment pre- and post-condition pairs are to specify the effect of an executable action on its environment. If the environment pre-condition of an executable action holds in the environment, then the corresponding environment post-condition of the action will be used to update the environment. If none of the environment pre-conditions of an executable basic action holds, then the agent can execute the action but the environment will not be updated. We assume that different environment pre-conditions of a basic action cannot be satisfied in one state.

In the following, we use $prec(\alpha)$ and $post(\alpha)$ to indicate the set of all pre-conditions and post-conditions of action α, respectively. Moreover, we use $eprec(\alpha)$, $bprec(\alpha)$, $epost(\alpha)$ and $bpost(\alpha)$ to denote the environment pre-conditions, the belief pre-conditions, the environment post-conditions, and the belief post-conditions of α, respectively. Note that the same action may have different pre-conditions and/or effects when executed by different agents. For example, a humanoid robot must have its hands empty in order to *lift* an object, while an Aibo should rather have its mouth empty instead. We will add the agent's name as a subscript in functions $prec$, $post$, etc. if the agent is not clear from the context.

REMARK 1. *Separate pre/postconditions wrt the agent's beliefs and the state of the environment allow to represent situations when the agent's beliefs are not objectively true. Moreover, they allow to model scenarios where the agent's expectations and effects of an action are different. It is important since we do not presuppose any relationship between beliefs and the objective state of the world.*

In most existing agent programming languages, the internal and external effects of an agent's action are specified separately in the agent program and the environment program. Interference between both kinds of effects is handled by synchronization of the agent's and the environment's actions. However, we do not assume any synchronization mechanism in our execution semantics (see Section 4). Note also that our approach allows to specify the interaction between the agent and the environment more directly. For example, the sensing *action can be given via clauses* `[`p`]{}sense{`$f(p)$`}[]`*, where f defines the agent's sensing capabilities.*

2.3 Plans

A plan consists of basic actions composed by sequence, conditional choice and conditional iteration operators. The sequence operator `;` takes two plans as arguments and indicates that the first plan should be performed before the second plan. The conditional choice and conditional iteration operators allow branching and looping, and generate plans of the form `if` ϕ `then` $\{\pi_1\}$ `else` $\{\pi_2\}$ and `while` ϕ `do` $\{\pi\}$ respectively. The condition ϕ is evaluated with respect to the agent's current beliefs. For example, the following plan causes the agent to grasp block `b` and put it on block `c`:

```
grasp_b ; put_b_c
```

2.4 Planning Rules

Planning rules are used to select a plan based on current goals and beliefs. A planning rule consists of three parts: an (optional) goal query, a belief query, and the body of the rule. The goal query specifies the state to be achieved by the plan and the belief query is used to denote the state the plan is believed to be executable. Firing a planning rule causes the adoption of the plan which forms the body of the rule. For example, consider the following planning rule which states that "if the goal is to have block `b` on block `c` and it is believed that block `c` is free, then block `b` should be grasped and put on block `c`":

```
on_b_c <- free_c | grasp_b ; put_b_c
```

[1]Note that this means that we only model so called *achievement goals* here.

For simplicity, we assume that agents do not have initial plans, i.e., plans can only be generated during the program execution by planning rules.

2.5 Programming Language: Formal Syntax

The formal syntax of the simple multi-agent programming language is given below in EBNF notation. We assume a set of belief update actions and a set of propositions. We use $\langle literal \rangle$ to denote a literal and $\langle baction \rangle$ to denote the name of a belief update action.

$\langle MA\_Prog \rangle$ ::= $\langle Agnt\_Prog \rangle$ ("," $\langle Agnt\_Prog \rangle$)*
$\langle Agnt\_Prog \rangle$ ::= "Actions:" $\langle updatespecs \rangle$
| "Beliefs:" $\langle literals \rangle$
| "Goals": $\langle goals \rangle$
| "Rules:" $\langle rules \rangle$
$\langle updatespecs \rangle$ ::= $\langle updatespec \rangle$ ("," $\langle updatespec \rangle$)*
$\langle updatespec \rangle$ ::= "[" $\langle literals \rangle$ "]" "{" $\langle literals \rangle$ "}" $\langle baction \rangle$ "{" $\langle literals \rangle$ "}" "[" $\langle literals \rangle$ "]"
$\langle literals \rangle$ ::= $\langle literal \rangle$ ("," $\langle literal \rangle$)*
$\langle goals \rangle$ ::= $\langle goal \rangle$ ("," $\langle goal \rangle$)*
$\langle goal \rangle$ ::= $\langle literal \rangle$ | $\langle goal \rangle$ "and" $\langle goal \rangle$
$\langle plan \rangle$ ::= $\langle baction \rangle$ | $\langle sequenceplan \rangle$
| $\langle ifplan \rangle$ | $\langle whileplan \rangle$
$\langle query \rangle$ ::= $\langle literal \rangle$
| $\langle query \rangle$ "and" $\langle query \rangle$
| $\langle query \rangle$ "or" $\langle query \rangle$
$\langle sequenceplan \rangle$ ::= $\langle plan \rangle$ ";" $\langle plan \rangle$
$\langle ifplan \rangle$ ::= "if" $\langle query \rangle$ "then {" $\langle plan \rangle$ "}" ["else {" $\langle plan \rangle$ "}"]
$\langle whileplan \rangle$ ::= "while" $\langle query \rangle$ "do {" $\langle plan \rangle$ "}"
$\langle rules \rangle$ ::= $\langle rule \rangle$ ("," $\langle rule \rangle$)*
$\langle rule \rangle$ ::= [$\langle query \rangle$] "<-" $\langle query \rangle$ "|" $\langle plan \rangle$

In the rest of the paper, we ignore $\langle ifplan \rangle$ and $\langle whileplan \rangle$ constructs because of the space limitation. They can be integrated in the framework in a fairly easy way.

2.6 BlocksWorld Example

In order to reduce the size of the example, we use `G_a` for `grasp_a` (grasp block `a`), `P_b_c` for `put_b_c` (put block `b` on `c`), `fr_c` for `free_c` (block `c` is free), and `c` for `carry` (carry a block). The multi-agent program $Blocks$ includes the following implementation of agent Ag_1:

```
Actions:
  [-on_d_b]{-c} G_b   {c}       []
  [-on_b_c]{c} P_b_c {-c,-fr_c}[on_b_c]
Beliefs:
  fr_c , -c
Goals:
  on_c_a and on_b_c and on_d_b
Rules:
  on_b_c <- fr_c | G_b ; P_b_c
```

and the following program implementing agent Ag_2:

```
Actions:
  [-on_b_c]{-c} G_c   {c}       []
  []        {-c} G_d   {c}       []
  [-on_c_a]{ c} P_c_a {-c,-fr_a}[on_c_a]
  [-on_d_b]{ c} P_d_b {-c,-fr_b}[on_d_b]
Beliefs:
  fr_a , fr_b , -c
Goals:
  on_c_a and on_b_c and on_d_b
Rules:
  on_d_b <- fr_b | G_d ; P_d_b
  on_c_a <- fr_a | G_c ; P_c_a
```

In this example, both agents have the same goal, i.e., to have block `d` on `b`, `b` on `c`, and `c` on `a`. Agent Ag_1 initially believes that block `c` is free and that it is not carrying a block. The agent is only capable of grasping block `b` (e.g., because of its arm capability and the shape of `b`) and putting it on block `c`, and it can generate a plan to have block `b` on `c`. Agent Ag_2 initially believes that blocks `a` and `b` are free and that it is not carrying a block. It has capability of grasping blocks `c` and `d` and putting them on other blocks. Moreover, it can generate plans to get blocks `d` on `b` and `c` on `a`. The execution of the resulting multi-agent program will be explained in Section 4 in more detail.

3. CHOOSING LOGIC AND SEMANTICS

There are many formal frameworks that can be used for modeling, reasoning about, and verification of multi-agent programs. In this section, we enumerate some of the options, and justify our choices for the rest of the paper. Some of the choices are tentative; we plan to explore other possibilities in future work.

First, let us discuss the **semantics of program execution**. Three major options are:

1. Synchronous models: each transition corresponds to simultaneous execution of actions by all the agents,
2. Asynchronous models with interleaving: each transition corresponds to an action executed by a single agent; action sequences from multiple agents are interleaved,
3. Asynchronous models with interleaving and synchronization by common action names.

Since the formal semantics of most BDI-based agent programming languages has been already based on the asynchrony assumption, and our proposed programming language does not include synchronization, we choose option 2. However, both other options are very interesting, and we plan to study them in the future.

Fairness assumptions are a closely related issue. We do *not* presuppose fairness of the program execution. However, we will show how fairness conditions can be specified directly in formulae of the logic is expressive enough, and how one can reason about properties of fair executions of programs (cf. Section 5.3).

Components of a system. Do we need a model of the external world (shared environment of action)? And, if so, what is the required relationship between agents' beliefs and the actual properties of the environment? We consider the following possibilities:

1. No environment, agents' beliefs are correct by definition,
2. With environment, beliefs are correct by definition,
3. With environment, beliefs can be correct or not.

The first option is sometimes used in frameworks for programming single agents, but for a multi-agent program we need a medium for agents' interaction. Moreover, we must take into account incorrect beliefs for a very simple reason: there is no way of ensuring that the programmer has programmed agents' beliefs so that they are consistent with each other. Therefore we choose option 3.

The choice of **logic** is crucial for what we can express and what kind of reasoning it facilitates. There are many logics of computation that can be used:

1. *Dynamic logic* [14] used primarily for reasoning about the end result of actions or (sequential) programs;

2. A variety of *temporal logics* based on the linear (LTL) and branching models of time (CTL), or logics that embed both perspectives (CTL*) [12];
3. *Strategic logics* with various degrees of expressivity: coalition logic [20], alternating-time temporal logic ATL and its more expressive variant ATL* [2]. Variants of the *stit* logic [4] also fall into this category;
4. The above logics can be combined with epistemic (resp. doxastic) logic when the agents' knowledge (resp. beliefs) are important. The combination is non-trivial especially in the case of strategic logics (cf. [17] for details).

In this work, we want to reason about what temporal patterns of execution can be triggered by which agent(s), hence the choice of ATL/ATL* seems most natural (it is much more expressive than coalition logic, whereas *stit* has a very complicated semantics and no immediate computational flavor). Beliefs are important for our setting, but our agents are not appropriate subjects of modal epistemic/doxastic logic.Thus, doxastic operators Bel_i are applied to propositional formulae only (the same holds for reasoning about goals).

Models of the logic: the action/time structure follows in a natural way from the operational semantics of the programming language presented in Section 4. Essentially, we deal with *labeled transition systems* with nodes representing global states of the system, and transitions labeled with the action that generates the transition and the agent that executes the action. Global system states combine local states of all the components. Moreover, in order to define the set of agents' strategies appropriately, we use an implicit epistemic structure by assuming that each agent can observe only its own local state (and thus a strategy of agent i must specify the same choices in global states that share the local state of i). In this sense, our models come very close to *interpreted systems* [13].

Finally, an important semantic choice concerns **capabilities of agents** (perfect vs. imperfect information, perfect vs. imperfect memory/recall, cf. [22]). Since we explicitly model agents' *beliefs* about the current state of the world, it does not make sense to assume perfect information (otherwise beliefs are identical with the current state of the environment). Moreover, the belief base is assumed to encapsulate all that the agent knows (or thinks) about the world, which corresponds to the notion of memoryless agents (i.e., ones that have no extra memory outside their current state).

In the following sections, we present an implementation of the choices outlined above.

4. OPERATIONAL SEMANTICS

We define the formal semantics of the agent programming language in terms of a transition system. Each transition corresponds to a single execution step and takes the system from one configuration to another. Configurations consist of the beliefs, goals, and plans of the agent. Which transitions are possible in a configuration depends on the agent program execution strategy. We have chosen asynchronous semantics of program execution, which means that transitions of different agents are interleaved rather than executed synchronously. Still, the scope of interleaving can be defined in at least two different ways. We can assume that each action (even a complex one, i.e., a plan) is treated as a whole and executed without interruption from another agent, or we can allow for interleaving of composite actions. Likewise, two execution strategies are also possible for execution of multiple plans by a *single* agent: one where the selected plan is executed to completion before choosing another plan, and another which interleaves the execution of multiple plans with the adoption of new plans. Out of the four combinations, we consider the two extremes here: complete execution of plans on both inter- and intra-agent levels vs. interleaving of plans on both inter- and intra-agent levels.

4.1 Configurations

A local state of an agent consists of the current beliefs, goals, and plans of the agent. A global state of program execution collects the current local states of all agents, plus the current state of the environment.

DEFINITION 1. *The configuration of a multi-agent program is defined as $\langle A_1, \ldots, A_n, \chi\rangle$, where $A_i = \langle i, \sigma, \gamma, \Pi\rangle$ is the configuration of agent i and χ is the state of the shared environment. In A_i, i is the agent's identifier, σ is a set of literals representing the agent's beliefs, γ is a set of conjunctions of literals representing the agent's goals, and Π is a set of plan entries representing the agent's current active plans.*

For example, the initial configuration of the two agents presented in section 2.6 is $\langle Ag_1, Ag_2, \chi\rangle$, where
$Ag_1 = \langle 1, \{\texttt{fr\_c}, -\texttt{c}\}, \{on\_c\_a \wedge on\_b\_c \wedge on\_d\_b\}, \emptyset\rangle$,
$Ag_2 = \langle 2, \{\texttt{fr\_a}, \texttt{fr\_b}, -\texttt{c}\}, \{on\_c\_a \wedge on\_b\_c \wedge on\_d\_b\}, \emptyset\rangle$,
and $\chi = \{\ -on\_c\_a\ ,\ -on\_b\_c\ ,\ -on\_d\_b\ \}$.
Note that the blocks in the initial state of the environment do not comply with the agents' goals.

Executing a multi-agent program modifies its initial configuration in accordance with the transition rules presented below.

4.2 Executing Actions of Different Agents

General Execution Rule. The following transition rule specifies the transition of multi-agent programs based on interleaving of the decisions generated by executing individual agents programs.

$$\frac{A_i \xrightarrow{\pi!} A_i' \quad \chi \xrightarrow{i:\pi?} \chi'}{\langle A_1, \ldots, A_i, \ldots, A_n, \chi\rangle \xrightarrow{i:\pi} \langle A_1, \ldots, A_i', \ldots, A_n, \chi'\rangle}$$

The condition of this transition rule indicates that the individual agent program i generates decision π in state A_i (changing the state to A_i') and the environment realizes the effect of that decision in its state χ (changing the state to χ'). The actual interleaving strategy depends on the way individual actions can be selected, cf. Section 4.3.

Execution of Shared Environment. Let i be an agent, α be a basic action and χ be a state of the environment. We assume that an environment update function for basic actions $UpdateE(i, \alpha, \chi) = \chi'$ is given beforehand. In this paper, we use an update function that consists in adding and removing atoms from the environment specified by the appropriate pre- and post-conditions:

$$UpdateE(i, \alpha, \chi) = \begin{cases} (\chi \setminus neg(epost_i(\alpha))) \cup pos(epost_i(\alpha)) \\ \qquad \text{if } \chi \models \bigwedge eprec(\alpha) \\ \chi \quad \text{otherwise} \end{cases}$$

where $neg(X) = \{p \mid -p \in X\}$ and $pos(X) = \{p \mid p \in X\}$ for each set of literals X, and $\models$ is the propositional entailment relation. The update function is extended to complex actions (plans) as follows:

$$UpdateE(i, \alpha; \pi, \chi) = UpdateE(i, \pi, UpdateE(i, \alpha, \chi))$$

The following transition rule specifies the effect of agent i's actions in the shared environment:

$$\frac{UpdateE(i, \pi, \chi) = \chi'}{\chi \xrightarrow{i:\pi?} \chi'}$$

The multi-agent program presented in Section 2.6 has several possible executions, among which the following: Ag_2 performs plan `G_d;P_d_b` followed by plan `G_c;P_c_a`, after which Ag_1 performs plan `G_b;P_b_c`. Note that the last plan performed by Ag_1 will be blocked. Another execution is one where Ag_2 first performs plan `G_c;P_c_a`, then agent Ag_1 performs `G_b;P_b_c`, and finally agent Ag_2 performs `G_d;P_d_b`. This execution achieves the agents' desired configuration of the blocks.

4.3 Execution of Individual Agent Programs

The execution of an individual agent program generates decisions based on its beliefs, goals, and planning rules. In this paper, we consider two execution strategies. In the first one, a planning rule is applied only if there is no plan to execute. The execution of a plan is atomic in the sense that different plans cannot be generated and executed in the interleaving mode. In the second strategy, different planning rules can be applied before plans can be executed. Moreover, different plans are executed in the interleaving mode.

Non-interleaving Execution Strategy. The effect of executing a basic action for an agent program state is that it updates the beliefs and goals. A basic action α can be executed if its precondition is entailed by the agent's beliefs, i,e., $\sigma \models \phi$. Executing the action adds the positive literals of its post-condition to the agent's beliefs and removes atoms of the negative literals. The belief update function $UpdateB(\alpha, \sigma)$ is formally defined based on belief pre- and post-condition of the basic action as follows:

$$UpdateB(\alpha,\sigma) = \begin{cases} (\sigma \setminus neg(bpost(\alpha))) \cup pos(bpost(\alpha)) \\ \qquad\qquad \text{if } \sigma \models \bigwedge bprec(\alpha) \\ UpdateB(\alpha,\sigma) = \bot \qquad \text{otherwise} \end{cases}$$

$$UpdateB(\alpha;\pi,\sigma) = UpdateB(\pi, UpdateB(\alpha,\sigma)).$$

The following transition rule specifies the effect of the execution of plan π on the state of an individual agent program.

$$\frac{UpdateB(\pi,\sigma) = \sigma' \neq \bot \quad \gamma' = \gamma \setminus \{\phi \in \gamma \mid \sigma' \models \phi\}}{\langle \sigma, \gamma, \{\pi\}\rangle \xrightarrow{\pi!} \langle \sigma', \gamma', \{\}\rangle}$$

Note that the goals are changed as a consequence of the belief update. In fact, the goals that are derivable from the updated beliefs are considered achieved and removed from the set of goals.

The transition rules for composite plans by sequence and conditional choice and loops can be defined in standard way. It is only important to note that the condition of the conditional choice and loops are evaluated with respect to the agent's beliefs.

In order to generate plans, planning rules should be applied. Let R be the set of planning rules of an agent program. The following transition rule specifies the application of a planning rule.

$$\frac{(\kappa \leftarrow \beta \mid \pi) \in R \quad \phi \in \gamma \quad \phi \models \kappa \quad \sigma \models \beta}{\langle \sigma, \gamma, \{\}\rangle \rightarrow \langle \sigma', \gamma', \{\pi\}\rangle}$$

Note that this transition rule does not allow the application of more than one planning rule as it requires that the set of plans should be empty before a planning rule can be applied.

As an example, consider agent program Ag_2 from Section 2.6. Non-interleaving execution of this agent program does not allow for executions where `G_c` follows directly after `G_d` (or vice versa).

Interleaving Execution Strategy. In this execution strategy, only atomic actions are executed so that actions of different plans can be interleaved. Note that this execution strategy makes it possible that different plans of different agents are executed in the interleaving mode. Let π be a plan (including the empty plan). The interleaving execution strategy is formally defined by the following transition rule:

$$\frac{\alpha;\pi \in \Pi \quad UpdateB(\alpha,\sigma) = \sigma' \neq \bot \quad \gamma' = \gamma \setminus \{\phi \in \gamma \mid \sigma' \models \phi\}}{\langle \sigma, \gamma, \Pi\rangle \xrightarrow{\alpha!} \langle \sigma', \gamma', (\Pi \setminus \{\alpha;\pi\}) \cup \{\pi\}\rangle}$$

In order to generate more plans so that their execution can be interleaved, we need to allow the application of more than one planning rule. This is done by the following transition rule (again, R is the set of planning rules of the agent program):

$$\frac{(\kappa \leftarrow \beta \mid \pi) \in R \quad \phi \in \gamma \quad \phi \models \kappa \quad \sigma \models \beta}{\langle \sigma, \gamma, \Pi\rangle \rightarrow \langle \sigma', \gamma', \Pi \cup \{\pi\}\rangle}$$

Note that the set of plans is not required to be empty now. Also, a planning rule can be applied several times. If a language designer considers such behavior undesirable, it can be avoided by adding additional constraints to the above transition rule. We do not discuss the issue further as it is not relevant for this paper.

5. STRATEGIC LOGIC FOR PROGRAMS

In order to reason about multi-agent programs, we use a variant of alternating-time temporal logic ATL [2]. ATL generalizes the branching time logic CTL [12] by replacing path quantifiers E, A with *cooperation modalities* $\langle\langle A\rangle\rangle$. Informally, $\langle\langle A\rangle\rangle\gamma$ expresses that the group of agents A have a collective strategy to enforce temporal property γ. ATL formulae include temporal operators: "$\bigcirc$" ("in the next state"), "$\Box$" ("always from now on"), "$\Diamond$" ("now or sometime in the future"), $\mathcal{U}$ (strong "until"), and $\mathcal{R}$ ("release").

For example, we can use the formula $\langle\langle\{1\}\rangle\rangle\Diamond\mathsf{win}$ to specify that agent 1 can eventually achieve a winning position in the game regardless of what the other agents do, how actions are interleaved, and so on. Moreover, formula $\langle\langle\{1,2\}\rangle\rangle\Box\mathsf{safe}$ says that agents 1 and 2 have a collective strategy that infallibly maintains safety of the system. Besides liveness and safety properties, we can also specify agents' abilities wrt various kinds of fairness. For instance, $\langle\langle\{i\}\rangle\rangle\Box\Diamond\mathsf{access_i}$ says that agent i can secure access to some resources infinitely many times, while $\langle\langle\{i\}\rangle\rangle(\Box\Diamond\mathsf{access_i} \rightarrow \Box\mathsf{safe}$ expresses that *if* i is granted the access infinitely often then it can successfully maintain the safety of the system.

Throughout the rest of the paper, we will write $\langle\langle a_1,\ldots,a_r\rangle\rangle$ instead of $\langle\langle\{a_1,\ldots,a_r\}\rangle\rangle$ to simplify the notation.

It is important to note that our treatment of the alternating-time logic is non-standard. First, we assume that agents have imperfect information and imperfect recall (cf. the discussion in [22]), which limits their available strategies to *uniform memoryless strategies*. In this sense, our logic can be seen as constructive strategic logic CSL [17] without constructive epistemic operators. Second, we use asynchronous models to define the semantics, whereas only synchronous semantics has been used so far for ATL and its variants (aside from a short discussion in [2] on how asynchronous systems can be "simulated" by synchronous models). Third, we include statements about agents' beliefs and goals, but with a very limited scope: they can refer only to objective properties of the world, just like the agent programs presented in Section 2 can. We call the resulting logic ATL$_P$ ("ATL for Programs") for lack of a more ingenious idea.

5.1 Syntax

Let $\mathbb{A}\mathrm{gt}$ be the set of all agents, and Π the set of atomic propositions occurring in multi-agent programs. The language of ATL$_\mathrm{P}$ is formally defined by the following grammar:

$$\varphi_0 ::= p \mid \neg\varphi_0 \mid \varphi_0 \wedge \varphi_0,$$
$$\varphi ::= \varphi_0 \mid \mathsf{Bel}_i\varphi_0 \mid \mathsf{Goal}_i\varphi_0 \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle A\rangle\rangle\gamma,$$
$$\gamma ::= \varphi \mid \neg\gamma \mid \gamma \wedge \gamma \mid \bigcirc\gamma \mid \gamma\,\mathcal{U}\,\gamma,$$

where $i \in \mathbb{A}\text{gt}$ is an agent, $A \subseteq \mathbb{A}\text{gt}$ is a group of agents, and $p \in \Pi$ is an atomic proposition. We also define $\Diamond\gamma \equiv \top\,\mathcal{U}\,\gamma$, $\gamma_1\,\mathcal{R}\,\gamma_2 \equiv \neg(\neg\gamma_1)\,\mathcal{U}\,(\neg\gamma_2)$, and $\Box\gamma \equiv \bot\,\mathcal{R}\,\gamma$.

Formulae φ_0 are *propositional formulae* and refer to simple facts about the current state of the system. Formulae φ are called *state formulae* and refer to (possibly more complex) properties of states. Finally, γ are *path formulae* that describe temporal patterns of particular execution paths. Additionally, we call a path formula γ *simple* iff the temporal operators in γ are only applied to state subformulae (like in $p_1\,\mathcal{U}\,\langle\langle A\rangle\rangle\Box p_2$, but not in $(p_1 \vee \bigcirc p_1)\,\mathcal{U}\,p_2$). Moreover, a (state or path) formula is *flat* if it does not include nested cooperation modalities.

Variants of the Logic. As it is often the case, there is a tradeoff between expressivity and complexity of decision procedures based on the logic. We consider several variants of ATL$_{\text{P}}$ (in accordance with the CTL and ATL tradition, cf. e.g. [21]). The full language is denoted by ATL$^*_{\text{P}}$, whereas a severely restricted version in which every occurrence of a cooperation modality is immediately followed by exactly one temporal operator is called "vanilla" ATL$_{\text{P}}$ (or simply ATL$_{\text{P}}$ if it does not cause any confusion).[2] "Vanilla" ATL$_{\text{P}}$ allows to express liveness and safety properties (as the examples at the beginning of Section 5 have already demonstrated), and it is cheaper than ATL$^*_{\text{P}}$ in terms of verification complexity (cf. Section 6.1). On the other hand, it does not allow to specify fairness conditions, which makes it useless for most scenarios (see the discussion in Section 5.3). The "Extended ATL$_{\text{P}}$" (EATL$_{\text{P}}$) augments "vanilla" ATL$_{\text{P}}$ with an additional temporal primitive $\Box\Diamond$ ("infinitely often"). Finally EATL$^+_{\text{P}}$ allows cooperation modalities to be followed by Boolean combinations of simple path formulae (with the primitive temporal operators being $\mathcal{U}$, $\Box\Diamond$).

5.2 Semantics

Models. We use the asynchronous labeled transition systems constructed in Section 4 as models of ATL$_{\text{P}}$. We recall that states of a system correspond to combinations $\langle A_1, \ldots, A_n, \chi\rangle$ of agent configurations $A_i = \langle i, \sigma(A_i), \gamma(A_i), \Pi(A_i)\rangle$ of individual agents (where $\sigma(A_i), \gamma(A_i)$, and $\Pi(A_i)$ are the beliefs, goals, and plans of agent i in configuration A_i), and the state χ of the shared environment. Transitions are labeled by $i{:}\alpha$ where α is an action and i is the agent that executes the action. We denote the set of i's configurations by St_i, the set of environment states by St_E, and the set of all global states of the system by $St \subseteq St_1 \times \cdots \times St_n \times St_E$. Likewise, for a global state $q \in St$, q_i will denote the $i's$ configuration in q, and q_E the state of the environment in q. The set of all actions occurring in transition labels is denoted by Act.

A *pointed model* (M, q) is a model together with a state in it. In the rest of the paper, we will use $pmodel(P)$ to denote the pointed model (M_P, q_0) that consists of the transition system M_P given by the operational semantics of multi-agent program P plus state q_o in M_P that corresponds to the initial configuration of P.

Truth of Formulae. In order to present semantic clauses for formulae of the logic, we need to first define the notions of a strategy and its outcome. A *strategy* of agent i specifies an executable action of i for each i's configuration. We will represent strategies of agent i by functions $s_i : St_i \to Act$ such that $\sigma(A_i) \models bprec_i(s_i(A_i))$. A *collective strategy* for a group of agents $A = \{a_1, \ldots, a_r\} \subseteq \mathbb{A}\text{gt}$ is simply a tuple $s_A = \langle s_{a_1}, \ldots, s_{a_r}\rangle$ of strategies, one per agent from A.

We define a path as a *full*[3] sequence of states interleaved with transitions. For path λ, we will denote the jth state on λ by $\lambda^{st}[j]$ and the jth transition on λ by $\lambda^{act}[j]$. $\lambda[j,\infty]$ denotes the part of λ from state number j onwards. Function $out(q, s_A)$ returns the set of all paths that may occur when agents A execute strategy s_A starting from state q:

$out(q, s_A) = \{\lambda = q_0(i_0{:}\alpha_0)q_1(i_1{:}\alpha_1)q_2 \ldots \mid q_0 = q$ and for each $j = 0, 1, \ldots$ there is a transition from $\lambda^{st}[j]$ to $\lambda^{st}[j+1]$ labeled with $\lambda^{act}[j] = i{:}\alpha$, and if $i \in A$ then $\alpha = s_i(\lambda^{st}[j]_i)\}$.

The semantics of ATL$^*_{\text{P}}$ can be now given by the following clauses:

$M, q \models p$ iff $\lambda^{st}[0]_E \models p$;

$M, q \models \mathsf{Bel}_i\varphi_0$ iff $\sigma(\lambda^{st}[0]_i) \models \varphi_0$;

$M, q \models \mathsf{Goal}_i\varphi_0$ iff $g \models \varphi_0$ for some $g \in \gamma(\lambda^{st}[0]_i)$;

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

$M, q \models \varphi \wedge \psi$ iff $M, q \models \varphi$ and $M, q \models \psi$;

$M, q \models \langle\langle A\rangle\rangle\gamma$ iff there is a collective strategy s_A for agents A such that for each path $\lambda \in out(s_A, q)$, we have $M, \lambda \models \gamma$;

$M, \lambda \models \varphi$ iff $M, \lambda^{st}[0] \models \varphi$;

$M, \lambda \models \neg\gamma$ iff $M, \lambda \not\models \gamma$;

$M, \lambda \models \gamma_1 \wedge \gamma_2$ iff $M, \lambda \models \gamma_1$ and $M, \lambda \models \gamma_2$;

$M, \lambda \models \bigcirc\gamma$ iff $M, \lambda[1,\infty] \models \gamma$; and

$M, \lambda \models \gamma_1\,\mathcal{U}\,\gamma_2$ iff there is $k \in \mathbb{N}_0$ such that $M, \lambda[k,\infty] \models \gamma_2$ and $M, \lambda[j,\infty] \models \gamma_1$ for all $0 \le j < k$.

For instance, formula $\langle\langle Ag_1\rangle\rangle\Diamond\mathsf{tower}$, where $\mathsf{tower} \equiv \mathsf{on\_d\_b} \wedge \mathsf{on\_b\_c} \wedge \mathsf{on\_c\_a}$, does *not* hold for the blocksworld program in Section 2.6. More formally, $pmodel(Blocks) \models \neg\langle\langle Ag_1\rangle\rangle\Diamond\mathsf{tower}$: agent Ag_1 cannot build the tower on its own.

Additional Remarks. We observe that $\langle\langle A\rangle\rangle\gamma$ does *not* mean that the agents in A *know how to play* to enforce γ (they cannot, since each $a \in A$ does not even have to be aware of existence of the other agents). It only means that A have an executable collective strategy so that if they execute it then γ will be the case.

Note also that the universal path quantifier A ("for every path") of CTL can be expressed with $\langle\langle\emptyset\rangle\rangle$ in ATL$_{\text{P}}$. Unlike in the original ATL, the existential path quantifier E can*not* be expressed with $\langle\langle\mathbb{A}\text{gt}\rangle\rangle$: first, memoryless strategies can yield only paths where agents periodically repeat their choices; second, whether a particular path will be obtained depends not only on the agents' choices but also on the order in which their actions will be scheduled by the execution platform (e.g., the Java Virtual Machine). On the other hand, "there is a path" properties can be expressed as $\mathsf{E}\gamma \equiv \neg\mathsf{A}\neg\gamma \equiv \neg\langle\langle\emptyset\rangle\rangle\neg\gamma$. Since all the considered variants of ATL$_{\text{P}}$ are closed under negation (modulo equivalence of formulae), $\neg\gamma$ can be always transformed into an appropriate form.

5.3 Examples. Fairness Badly Needed!

Consider the blocksworld example from Section 2.6 and the property $\mathsf{tower} \equiv \mathsf{on\_d\_b} \wedge \mathsf{on\_b\_c} \wedge \mathsf{on\_c\_a}$. We already mentioned that $pmodel(Blocks) \models \neg\langle\langle Ag_1\rangle\rangle\Diamond\mathsf{tower}$. The other agent is in no better position: $pmodel(Blocks) \models \neg\langle\langle Ag_2\rangle\rangle\Diamond\mathsf{tower}$. However, the agents have a collective strategy to build the tower together: $pmodel(Blocks) \models \langle\langle Ag_1, Ag_2\rangle\rangle\Diamond\mathsf{tower}$. Can any agent achieve something on its own? Yes, for example we have that $pmodel(Blocks) \models \langle\langle Ag_1\rangle\rangle\Diamond\mathsf{Bel}_{Ag_1}\mathsf{carry}_1$. But that is only because the agents in $Blocks$ have been programmed so that the behavior of the system is almost deterministic. In particular, agent

[2] In "vanilla" ATL$_{\text{P}}$, the "release" operator $\mathcal{R}$ is added as another primitive, since it cannot be expressed with $\mathcal{U}$ any more.

[3] I.e., either infinite or ending in a deadlock state

Ag_2 must eventually come to a blocking point where it has to wait for some action of Ag_1.

Suppose, on the other hand, that we modify the programs of Ag_1, Ag_2 so that the agents can freely grab and put down "their" blocks whenever those are available. Then, $\langle\langle Ag_1\rangle\rangle\Diamond \mathsf{Bel}_{Ag_1}\mathsf{carry}_1$ does not hold any more: the execution platform can postpone Ag_1's actions forever! More generally, agents' achievement abilities are very limited without fairness assumptions, as the following proposition demonstrates.

PROPOSITION 1. *For every multi-agent program P and ATL$_P$ formula φ there is an agent i in P such that, for every coalition $A \subseteq \mathbb{A}\mathrm{gt} \setminus \{i\}$, we have $pmodel(P) \models \mathsf{E}\Box\neg\varphi \rightarrow \neg\langle\langle A\rangle\rangle\Diamond\varphi$.*

How can we express fairness? We suggest the following construction. First, we augment programs with additional atoms act_i, one per agent. The atoms are supposed to keep track of the most recent actor. That is, the environment postconditions are modified as follows: $epost'_i(\alpha) = epost_i(\alpha) \cup \{\mathsf{act}_i\} \cup \{-\mathsf{act}_j \mid j \neq i\}$. Now, for example, the ATEL$_P$ formula $\langle\langle c\rangle\rangle\Box\Diamond\mathsf{act}_i$ specifies that there is a "controller" agent c who can enforce that only computations fair for agent i are executed. But that is still not enough to express fairness with respect to all agents, and to reason about what can be achieved *if we take only fair computations into account*.

As it turns out, ATEL^+_P is sufficient for such properties. Consider the path formula $\mathsf{fair} \equiv \bigwedge_i \Box\Diamond\mathsf{act}_i$ which expresses that, at no future moment, an agent may be prevented from acting forever. The ATEL^+_P formula $\langle\langle Ag_1\rangle\rangle(\mathsf{fair} \rightarrow \Diamond\mathsf{Bel}_{Ag_1}\mathsf{carry}_1)$ says that Ag_1 has a strategy to *achieve* $\mathsf{Bel}_{Ag_1}\mathsf{carry}_1$ *for every fair computation*, which is indeed true for our modified blocksworld scenario.

6. VERIFICATION

Let P be a multi-agent program, (M_P, q_0) the pointed model of P, and φ a state formula of ATL$^*_P$. The model checking problem for P, φ asks whether $M_P, q_0 \models \varphi$. Thus, it is the decision problem that takes either a program or its more extensive representation (as a transition system), and a logical specification whose truth value should be determined. In Section 6.1, we discuss the theoretical complexity of model checking for different variants of ATL$_P$. Unsurprisingly, the problem is not easy; we make the first step towards decreasing the complexity – at least for some programs – in Section 6.2. We omit proofs due to lack of space; an interested reader is referred to the technical report [11].

6.1 Complexity of Model Checking

Complexity of decision problems is always studied in relation to the size of problem instances. In our case, this means that we investigate the complexity of model checking wrt the length of the formula and the representation of the multi-agent program P, i.e., either the program itself or its model M. We give complexity results for both kinds of analysis.

THEOREM 1. *Model checking ATL$^*_P$, EATL^+_P, EATL$_P$, and "vanilla" ATL$_P$ is* **PSPACE***-complete with respect to the size of the program and the length of the formula.*

That is, the complexity of verification is rather prohibitive. This, however, is not due to our choice of logic: the same complexity results have been obtained for model checking "bare" temporal logics LTL, CTL, and CTL* [21]). The complexity becomes a little more optimistic when measured in terms of the size of the *model* rather than the *program*. One should be cautious, however: the size of the model is usually exponential in the original program!

THEOREM 2. *Model checking ATL$^*_P$ is* $\mathbf{PSPACE}$*-complete with respect to the number of transitions in M and the length of φ.*

THEOREM 3. *Model checking "vanilla" ATL$_P$ and EATL$_P$ is* $\mathbf{\Delta_2^P}$*-complete wrt the no. of transitions in M and the length of φ.*

THEOREM 4. *Model checking EATL^+_P is* $\mathbf{\Delta_3^P}$*-complete with respect to the number of transitions in M and the length of φ.*

According to the above results, ATEL^+_P seems a sensible choice: it is expressive enough to capture most interesting properties of multi-agent programs, but still distinctly cheaper than the full ATL$^*_P$ in terms of verification. However, complexity wrt the size of the model often "hides" the exponential blowup that occurs during model generation. Several techniques have been proposed to reduce the complexity of models and facilitate model checking – most notably, *abstraction* and *partial order reduction*. We make the first step towards the latter kind of reduction in the next section.

6.2 Partial-Order Reduction

For an asynchronous agent system, the blowup in the size of the model is partially due to exponentially many interleavings of agents' actions. *Partial order reduction* [8] is a well-known technique used to reduce the number of interleavings (and hence also global states in the model) that must be taken into account. The main idea behind the reduction is to collapse paths that differ only in the ordering of mutually independent actions. This independence is captured formally by an appropriate notion of *stuttering equivalence* between paths and models. In this section, we define a stuttering equivalence for a subset of ATL$^*_P$ that includes most (if not all) formulae relevant for verification.

DEFINITION 2 (ATL$^{\oslash}_P$). *The language of ATL$^{\oslash}_P$ consists of all the flat formulae of ATL$^*_P$ that include no $\bigcirc$ operator.*

DEFINITION 3 (A-STRATEGIC EQUIVALENCE). *Let $P \subseteq \Pi$ be a (sub)set of propositions, and let P_i^B (resp. P_i^G) collect propositions that can appear in agent i's belief (resp. goal) base. We define $labels^P(\lambda)$ as the sequence of: (1) valuations of propositions from P in states on λ, (2) valuations of propositions from $P_i^B \cap P$ in i's belief states on λ, and (3) valuations of propositions from $P_i^G \cap P$ in i's goal states on λ, with subsequent identical entries collapsed into a single entry. We also extend $labels_i^P(\lambda)$ to $labels_A^P(\lambda)$ in a natural way.*

*We say that λ is A-*stuttering equivalent to λ' wrt P *(written $\lambda \cong_A^P \lambda'$) iff $labels_A^P(\lambda) = labels_A^P(\lambda')$. Pointed models (M, q_0), (M', q'_0) are A-*strategically equivalent wrt P *(written $(M, q_0) \cong_A^P (M', q'_0)$) iff for every s_A in M there is s'_A in M' with $out_M(q_0, s_A) \cong_A^P out_{M'}(q'_0, s'_A)$, and vice versa.*

The following is a straightforward consequence of the result in [8].

THEOREM 5. *If $(M, q_0) \cong_A^P (M', q'_0)$ then for every ATL$^{\oslash}_P$ formula φ that includes only agents from A and propositions from P, we have that $M, q_0 \models \varphi$ iff $M', q'_0 \models \varphi$.*

Defining an appropriate stuttering equivalence is the first (and most important) step in adapting partial order reduction to our logic. Then, one can use the equivalence to define a heuristics for model (re)construction. We leave the second part for future work, and refer to [8] for the general idea.

6.3 Stuttering Bisimulation

Unfortunately, the notion of strategic equivalence proposed in Section 6.2 is only useful when the number of available strategies is relatively small. In this section, we briefly outline a notion of equivalence between program models, that can be used alternatively.

DEFINITION 4 (A-BISIMULATION). *Pointed models* (M_1, q_1), (M_2, q_2) *are* A-bisimilar wrt P *(written* $(M_1, q_1) \sim_A^P (M_2, q_2)$*) iff: (1)* q_1, q_2 *agree on* A*'s beliefs and goals, and on the "objective" truth of propositions from* P*, (2) for every transition* $q_1 \xrightarrow{i:\alpha} q_1', i \in A$ *in* M_1 *there is* $q_2 \xrightarrow{i:\alpha} q_2'$ *in* M_2 *such that* $(M_1, q_1') \sim_A^P (M_2, q_2')$*, (3) for every transition* $q_1 \xrightarrow{r:\alpha} q_1', r \notin A$ *in* M_1 *there is* $q_2 \xrightarrow{r:\alpha} q_2'$ *in* M_2 *such that* $(M_1, q_1') \sim_A^P (M_2, q_2')$*, (4) and vice versa.*

DEFINITION 5 (STUTTERING A-BISIMULATION). *To define* stuttering A-bisimulation wrt P *(written* $(M_1, q_1) \approx_A^P (M_2, q_2)$*), we replace condition (3) in Definition 4 as follows: (3') for every transition sequence* $q_1 \xrightarrow{r_1:\alpha_1} \cdots \xrightarrow{r_n:\alpha_n} q_1', r_1, \ldots, r_n \notin A$ *in* M_1 *there is* $q_2 \xrightarrow{r_1':\alpha_1'} \cdots \xrightarrow{r_m':\alpha_m'} q_2', r_1', \ldots, r_m' \notin A$ *in* M_2 *such that the two sequences stutteringly agree on* A*'s beliefs,* A*'s goals, and propositions from* P*, and* $(M_1, q_1') \approx_A^P (M_2, q_2')$.

THEOREM 6.
If $(M_1, q_1) \sim_A^P (M_2, q_2)$ *then* $(M_1, q_1) \approx_A^P (M_2, q_2)$.

THEOREM 7.
If $(M_1, q_1) \approx_A^P (M_2, q_2)$ *then* $(M_1, q_1) \cong_A^P (M_2, q_2)$.

Thus, bisimulation is a special case of stuttering bisimulation, which is in turn a case of strategic equivalence. Note that the full notion of stuttering bisimulation allows us to merge all the labels of the opponents' actions into a single action label, collapse states that do not change A's position, and then remove sequences that display the same pattern. By Theorems 7 and 5, such a model reduction preserves the strategic abilities of coalition A, and hence also the truth of flat $\langle\langle A \rangle\rangle \gamma$ formulae.

7. CONCLUSIONS

In this paper, we present an adaptation of the strategic logic ATL to reasoning about multi-agent programs. We carefully outline possible choices regarding the model of execution, specification language, and semantics of formulae. Then, we use a variant of ATL with imperfect information and imperfect recall, and interpret its formulae over asynchronous labeled transition systems that arise from the operational semantics of the agent programming language. Initial complexity results for the model checking problem suggest that verification is feasible only for relatively small programs. Nevertheless, some techniques can be used in order to cut down the complexity in practice. We define an appropriate notion of stuttering equivalence that allows to reduce models of the logic where their blowup is caused by interleaving of independent actions. To our knowledge, it is the first attempt at applying a model reduction technique for a variant of alternating-time logic.

We also argue that fairness is an indispensable notion when reasoning about what agents can achieve in an asynchronous setting, and we show how it can be imposed in the object language of our logic. Taking into account the complexity and expressivity of different variants of the logic, it seems that the variant $\text{EATL}_{\text{P}}^{+}$ strikes the balance best: it enables to specify most interesting properties of agent programs while keeping the verification complexity distinctly lower than for the full $\text{ATL}_{\text{P}}^{*}$.

8. REFERENCES

[1] N. Alechina, M. Dastani, B. Logan, and J.-J. C. Meyer. A logic of agent programs. In *Proceedings of AAAI*, pages 795–800, 2007.

[2] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time Temporal Logic. *Journal of the ACM*, 49:672–713, 2002.

[3] L. Astefanoaei, M. Dastani, F. de Boer, and J.-J. C. Meyer. A verification framework for normative multi-agent systems. In *Proceedings of PRIMA 2008*, volume 5357 of *LNCS*. Springer, 2008.

[4] N. Belnap and M. Perloff. Seeing to it that: a canonical form for agentives. *Theoria*, 54:175–199, 1988.

[5] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multiagent systems. *Journal and Logic and Computation*, 8(3):401–423, 1998.

[6] R. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[7] R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. John Wiley & Sons, 2007.

[8] E. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1998.

[9] P. Cohen and H. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[10] M. Dastani. 2APL: a practical agent programming language. *International Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 16(3):214–248, 2008.

[11] M. Dastani and W. Jamroga. Reasoning about strategies of multi-agent programs. Technical Report IfI-10-04, Clausthal University of Technology, 2010.

[12] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.

[13] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[14] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[15] K. Hindriks. Modules as policy-based intentions: Modular agent programming in GOAL. In *Procedings of ProMAS'07*, volume 4908 of *LNCS*. Springer, 2008.

[16] K. V. Hindriks and J.-J. C. Meyer. Toward a programming theory for rational agents. *Autonomous Agents and Multi-Agent Systems*, 19(1):4–29, 2009.

[17] W. Jamroga and T. Ågotnes. Constructive knowledge: What agents can achieve under incomplete information. *Journal of Applied Non-Classical Logics*, 17(4):423–475, 2007.

[18] W. Jamroga and J. Dix. Model checking ATL_{ir} is indeed Δ_2^P-complete. In *Proceedings of EUMAS'06*, 2006.

[19] F. Laroussinie, N. Markey, and G. Oreiby. On the expressiveness and complexity of ATL. *LMCS*, 4:7, 2008.

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

[21] P. Schnoebelen. The complexity of temporal model checking. In *Advances in Modal Logics, Proceedings of AiML 2002*. World Scientific, 2003.

[22] P. Y. Schobbens. Alternating-time logic with imperfect recall. *Electronic Notes in Theoretical Computer Science*, 85(2), 2004.