Reasoning about Strategic Properties of Multi-Agent Programs

Mehdi Dastani[1] and Wojciech Jamroga[2,3]

[1] Institute of Information and Computer Sciences, Utrecht University, the Netherlands
[2] Computer Science and Communications, University of Luxembourg
[3] Department of Informatics, Clausthal University of Technology, Germany
mehdi@cs.uu.nl wojtek.jamroga@uni.lu

Abstract. Specification and verification of multi-agent programs is a key problem in agent research and development. One may for example be interested in checking if a specific subset of individual agent programs can achieve a particular state of their shared environment, or if they can protect the system from entering a "bad" state. This paper focuses on BDI-based 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 such properties. 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 of modal epistemic logic either (since they do not know the global model of the system). This paper presents our first attempt at adapting the semantics of ATL to reasoning about asynchronous multi-agent programs.

1 Introduction

Specification and verification of agent-based systems is a key problem in multi-agent research and development [13, 28]. More recently, there has been a shift focusing on the verification of *multi-agent programming languages* and *multi-agent programs*, i.e., programs that are developed to implement multi-agent systems [14, 20, 9]. These works aim at verifying safety and liveness properties in terms of the internals of individual agent programs such as beliefs, goals, and plans [8, 7, 1, 5]. Examples of such properties are realism, various kinds of commitment strategies, and communication abilities.

This paper concentrates on reasoning about (and verification of) BDI-based multi-agent programs that consist of a finite set of BDI-based individual agent programs operating in a shared environment and executed concurrently. We propose a framework that allows the developers of multi-agent programs to check if (a subset of) individual agent programs can cooperate to achieve a specific state of the shared environment. In particular, the paper focuses on reasoning about properties related to possible cooperations between individual agent programs. One may for example be interested in checking if a specific subset of individual agent programs, developed to operate in a blockworld environment, can cooperate to achieve a state where they have built a certain tower of blocks.

In order to develop such a framework, the logic used for reasoning about multi-agent programs must be grounded in the computation of these programs. In this paper, we first present an APL-like multi-agent programming language which is a simplified and modified version of an existing multi-agent programming language called 2APL [14]. It is simplified because we focus on ingredients that also appear in other APL-like programming languages (e.g., [20, 9]), and it is modified because all actions are now specified with respect to both agents' internals as well as the shared environment. We then present a logic for APL-like multi-agent programming languages, that can be used to reason about the execution of multi-agent programs and possible cooperations between agents in those programs. Among other things, the logic allows for specification and verification of safety and liveness properties related to the cooperation and strategic abilities of the agents.

We begin by presenting the multi-agent version of 2APL in Section 2. Then, we discuss the choice of a formalism for specification and verification of properties for 2APL 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, with semantics defined over labeled interpreted systems that arise from an asynchronous model of program execution. Sections 4 and 5 present the formal framework that implements the choices. That is, in Section 4 we present an operational semantics for 2APL that generates appropriate models of multi-agent programs, and in Section 5 we propose a variant of alternating-time temporal logic that seems suitable for reasoning about their properties. We also briefly discuss the verification problem for the setting: we establish preliminary complexity results and speculate how the complexity of the procedure can be reduced to make verification feasible.

2 A Programming Language for Multi-Agent Systems

In this section we present the syntax of a fragment of the BDI-based multi-agent programming language 2APL [14]. This fragment contains the core features of 2APL and allows the 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 BDI-based programs for individual agents and a shared environment in which all agents can perform actions. The shared environment is represented by a set of atoms. 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 necessary all at once).[4]

[4] Note that this means that we only deal with so called *achievement goals* here.

For simplicity, we represent an agent's beliefs as a set of atoms and goals as a set of literals. We also assume the closed world principle with respect to beliefs: the agent believes that $\neg p$ iff p is not in its current belief base. The initial beliefs and goals of an agent are specified by its program. 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).

2.2 Basic Actions

Basic actions specify the capabilities an agent has to achieve its goals. In this paper and in contrast to previous expositions of 2APL, we consider basic actions of an agent as being specified in terms of both the agent's beliefs as well as properties of the 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. One of the pre-conditions is a condition on the agent's beliefs and the second is condition on the environment. The pre- and post-conditions are sets of literals. One post-condition determines the update of the agent's beliefs and the second determines the update of the environment. For example, consider the specification of basic action `put_X_Y`, which consists of two tuples of pre- and post-conditions:

```
[-on_X_Y]{ free_Y}  put_X_Y  {-free_Y}[ on_X_Y]
[-on_X_Y]{-free_Y}  put_X_Y  { free_X}[floor_X]
```

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-condition of the first pair indicates that the action can be executed if the agent believes block `Y` is free. Its corresponding post-condition indicates that after the execution of the action the agent will believe block `Y` is not free anymore. The environment pre- and post-conditions indicate that the execution of the action will put block `X` on block `Y` if it was not already the case.

In general, a basic action can be executed only if one of its belief pre-conditions is derivable from the agent's beliefs (based on the closed world assumption as the agent's belief base is a set of atoms). 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. Basic actions maintain consistency of the agent's beliefs. We assume 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. This means that basic actions can be executed even if none of its environment pre-conditions holds in the environment. In the above example, if the agent believes `-freeY`, then the action can be executed and its effect will be determined based on the second pre- and post-condition pair. In particular, if block `X` was not on block `Y`, then the execution of the action results is a state where block `X` is on the floor. Otherwise the basic action can be executed without any environment effect. We assume that different environment pre-conditions of a basic action cannot be satisfied in one state. A basic action α is specified as follows:

$$\begin{aligned} \alpha = \{\ & \langle\ (eprec_1, bprec_1), (bpost_1, epost_1)\ \rangle \\ & \vdots \\ & \langle\ (eprec_n, bprec_n), (bpost_n, epost_n)\ \rangle \\ \} & \end{aligned}$$

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-condition, the belief pre-condition, the environment post-condition and belief post-condition of α, respectively. Thus:

$$\begin{aligned} prec(\alpha) &= \{\ (eprec_1, bprec_1), \ldots, (eprec_n, bprec_n)\ \} \\ post(\alpha) &= \{\ (epost_1, bpost_1), \ldots, (epost_n, bpost_n)\ \} \\ bprec(\alpha) &= \{bprec_1, \ldots, bprec_n\} \\ eprec(\alpha) &= \{eprec_1, \ldots, eprec_n\} \\ bpost(\alpha) &= \{bpost_1, \ldots, bpost_n\} \\ epost(\alpha) &= \{epost_1, \ldots, epost_n\} \end{aligned}$$

Note that the same action may have different preconditions 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. We also observe that the agent's perception can be encoded as the interplay between its environment pre-conditions and belief post-conditions. In particular, one can define action $perceive$ with the following specification:

```
[ p]{} perceive { p}[]
[-p]{} perceive {-p}[]
```

where p expresses an atomic property observable to the agent.

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 first put block `b` on `c`, and then put block `a` on `b`.

```
put_b_c;put_a_b
```

2.4 Planning Goal Rules

Planning goal rules are used to select a plan based on current goals and beliefs. A planning goal rule consists of three parts: an (optional) goal query, a belief query, and the body of the rule. The goal query specifies the state(s) to be achieved by the plan, and the belief query denotes the states in which the plan is believed to be executable. Firing a planning goal rule results in adoption of the plan which forms the body of the rule. For example, consider the following planning goal rules.

```
buildTower  <-  free_X and free_Y  |  grasp_X;put_X_Y
buildTower  <-  free_X and -free_y |  grasp_X;put_X_fl
```

The first rule states that "if the goal is to build a tower of blocks and it is believed that blocks X and Y are free, then block X should be grasped and put on block Y". A plan can also be generated independently of goals but based on current beliefs only. This allows the implementation of *reactive* agents (agents without any goals). For example, the following reactive planning rule.

```
  <- -battery | goToStation;chargeBattery
```

This rule states "if the battery is low, then go to the loading station and charge the battery". For simplicity, we assume that agents do not have initial plans, i.e., plans can only be generated during the program execution by planning goal rules.

The syntax of the programming language is given below in EBNF notation. We assume a set of belief update actions and a set of propositions, and use ⟨*atoms*⟩ to denote a sequence of atoms, ⟨*literal*⟩ to denote a literal, and ⟨*aliteral*⟩ to denote the name of a belief update action.

```
⟨APL_Prog⟩      ::= "Actions:" ⟨updatespecs⟩
                  | "Beliefs:" ⟨atoms⟩
                  | "Goals": ⟨literals⟩
                  | "PG rules:" ⟨pgrules⟩
⟨updatespecs⟩   ::= [⟨updatespec⟩ ("," ⟨updatespec⟩)* ]
⟨updatespec⟩    ::= "[" ⟨literals⟩ "]" "{" ⟨literals⟩ "}"
                                 ⟨aliteral⟩
                     "{"⟨literals⟩"}" "[" ⟨literals⟩ "]"
⟨literals⟩      ::= [⟨literal⟩ ("," ⟨literal⟩)*]
⟨plan⟩          ::= ⟨baction⟩ | ⟨sequenceplan⟩ | ⟨ifplan⟩ | ⟨whileplan⟩
⟨baction⟩       ::= ⟨aliteral⟩ | ⟨testbelief⟩ | ⟨testgoal⟩
⟨testbelief⟩    ::= ⟨bquery⟩ "?"
⟨testgoal⟩      ::= ⟨gquery⟩ "!"
⟨bquery⟩        ::= ⟨literal⟩ | ⟨bquery⟩ "and" ⟨bquery⟩ | ⟨bquery⟩ "or" ⟨bquery⟩
⟨gquery⟩        ::= ⟨literal⟩ | ⟨gquery⟩ "and" ⟨gquery⟩ | ⟨gquery⟩ "or" ⟨gquery⟩
⟨sequenceplan⟩  ::= ⟨plan⟩ ";" ⟨plan⟩
⟨ifplan⟩        ::= "if" ⟨bquery⟩ "then {" ⟨plan⟩ "}" ["else {" ⟨plan⟩ "}"]
⟨whileplan⟩     ::= "while" ⟨bquery⟩ "do {" ⟨plan⟩ "}"
⟨pgrules⟩       ::= [⟨pgrule⟩ ("," ⟨pgrule⟩)* ]
⟨pgrule⟩        ::= [⟨gquery⟩] "<-" ⟨bquery⟩ "|" ⟨plan⟩
```

3 Choosing Logic and Semantics

There are plenty of formal framework 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 the 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 based on 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 multi-agent 2APL has been already based on the asynchrony assumption, and the fragment of 2APL studied here 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. In this preliminary study, we do not assume fairness of the program execution.

Components of a system. Do we need a model of the external world, i.e., of the environment of action, shared by all agents? 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.

As we already suggested in Section 2, we choose option 3. 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.

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* [18] used primarily for reasoning about the end result of actions or (sequential) programs;
2. A variety of *temporal logics* based on the linear (LTL [27]) and branching model of time (CTL [10]), or logics that embed both perspectives (CTL* [15]);
3. Combinations of dynamic and temporal logic that allow to reason about temporal patterns that result from executing programs: Process Logic [17], DLTL [19], DCTL*[25];
4. *Strategic logics* with various degrees of expressivity: coalition logic [26], alternating-time temporal logic ATL [3, 4] and its more expressive variant ATL* [4]; variants of the *stit* logic [6] also fall into this category;

5. 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. [21] 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 since we cannot assume that they know the whole model of the system (they are not even aware of existence of other agents!). 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 consequence, we use multi-modal Kripke models that come very close to *interpreted systems* [16].

Finally, an important semantic choice concerns **capabilities of agents** (perfect vs. imperfect information, perfect vs. imperfect memory/recall, cf. [30, 24]). 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 redundant as 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 of Multi-Agent Programs

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, possibly also with 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 is the configuration of an individual agent and χ is the state of the shared environment. The configuration of an individual agent A_i is defined as $\langle i, \sigma, \gamma, \Pi\rangle$ where i is an agent identifier, σ is a set of atoms representing the agent's beliefs, γ is a set of literals representing the agent's goals, and Π is a set of plan entries representing the agent's current active plans.*

The initial beliefs and goals of individual agent are specified by their programs. The agents are assumed to have no initial plans. The initial state of the shared environment is assumed to be given by the system developer. 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.

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

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 $UpdateE(\alpha, \chi) = \chi'$ for basic actions 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:

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

where $neg(X) = \{p \mid -p \in X\}$ and $pos(X) = \{p \mid p \in X\}$ for each set of literals X, and $\models_{cwa}$ is the entailment relation based on closed-world assumption.

The update function is extended to complex actions (plans) as follows:

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

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

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

4.3 Execution of Individual Agent Programs

The execution of an individual agent program generates decisions based on the agent's beliefs, goals, and planning goal rules. There are various strategies to execute an individual agent program. For example, when in a configuration with no plan, choose a planning goal rule non-deterministically, apply it, execute the resulting plan; repeat.

In this paper, we consider two execution strategies. In the first execution strategy, one PG-rule is applied only if there is no plan to execute, i.e., a PG-rule is applied and its plan is executed before another PG-rule is applied. 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 PG-rules can be applied before plans can be executed. Moreover, different plans can be 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., $\sigma \models_{cwa} \phi$. Executing the action adds the positive literals of its post-condition to the agent's beliefs and removes atoms of the negative literals. Let $UpdateB(\alpha, \sigma) = \sigma'$ be a belief update function and $UpdateB(\alpha; \pi, \sigma) = UpdateB(\pi, updateB(\alpha, \sigma))$. This update function can be defined based on belief pre- and post-condition of the basic action as follows.

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

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

$$\frac{\pi \in \Pi \quad UpdateB(\pi, \sigma) = \sigma' \neq \bot \quad \gamma' = \gamma \setminus \{\phi \in \gamma \mid \sigma' \models_{cwa} \phi\}}{\langle \sigma, \gamma, \{\pi\} \rangle \overset{\pi!}{\rightarrow} \langle \sigma', \gamma', \{\} \rangle}$$

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

The transition rules for belief and goal test actions as well as for composite plans by sequence and conditional choice and loops can be defined in standard way [14].

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

$$\frac{\kappa \leftarrow \beta \mid \pi \in R \quad \kappa \in \gamma \quad \sigma \models_{cwa} \beta}{\langle \sigma, \gamma, \{\} \rangle \to \langle \sigma', \gamma', \{\pi\} \rangle}$$

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

Interleaving Execution Strategy. In this execution strategy, only atomic actions are executed such 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 (possibly empty). The execution strategy is defined by the following rule:

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

In order to generate more plans so that the execution of their actions can be interleaved, we need to allow the application of multiple PG-rules. This is done by the following transition rule. Let R be the set of PG-rules of an agent program.

$$\frac{\kappa \leftarrow \beta \mid \pi \in R \quad \kappa \in \gamma \quad \sigma \models_{cwa} \beta}{\langle \sigma, \gamma, \Pi \rangle \to \langle \sigma', \gamma', \Pi \cup \{\pi\} \rangle}$$

Note that the set of plans is not required to be empty.

5 Strategic Logic for Multi-Agent 2APL Programs

In order to reason about multi-agent programs, we use a variant of alternating-time temporal logic ATL [3, 4]. ATL generalizes the branching time logic CTL [10] by replacing path quantifiers E, A with so called *cooperation modalities* $\langle\langle A \rangle\rangle$. Informally, $\langle\langle A \rangle\rangle \gamma$ expresses that 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 $\mathcal{W}$ (weak "until").

Our treatment of the logic is non-standard, though. First, we assume that agents have imperfect information and imperfect recall, which limits their available strategies to *uniform memoryless* strategies. In this sense, our logic can be seen as constructive strategic logic CSL [21] without 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 discussion in [4] on how asynchronous systems can be "simulated" by synchronous models). Third, we include statements about agents' beliefs and goals, but they can refer only to objective properties of the world, just like in the syntax of 2APL. We call the resulting logic 2ATL 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 a multi-agent program. The language of 2ATL is formally defined by the following grammar:

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

where $i \in \mathbb{A}\mathrm{gt}$ is an agent, $A \subseteq \mathbb{A}\mathrm{gt}$ is a group of agents, and $p \in \Pi$ is an atomic proposition. We also define $\Diamond\gamma \equiv \top\,\mathcal{U}\,\gamma$. 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.

Example properties that can be expressed include: $\langle\!\langle 1,2\rangle\!\rangle\Diamond\mathsf{tower}$ (agents 1 and 2 have a joint strategy to build a tower eventually), $\langle\!\langle 1,2\rangle\!\rangle\Box\mathsf{clean}$ (agents 1 and 2 have a joint strategy to keep the floor always clean), and $\langle\!\langle 1,2\rangle\!\rangle(\Box\mathsf{clean} \wedge \Diamond\mathsf{tower})$: agents 1 and 2 have a joint strategy to to build a tower while keeping the floor clean all the time.

We consider two variants of 2ATL (in accordance with the CTL and ATL tradition). The full language is denoted by 2ATL*, whereas a restricted version in which every occurrence of a cooperation modality is immediately followed by exactly one temporal operator is called "vanilla" 2ATL (or simply 2ATL if it does not cause any confusion). Note that, for 2ATL*, operator $\Box$ is redundant since it can be defined as $\Box\gamma \equiv \neg\Diamond\neg\gamma$.

5.2 Semantics

We use the asynchronous labeled transition systems constructed in Section 4 as models of 2ATL. We recall that states of a system correspond to combinations $\langle A_1, \ldots, A_n, \chi\rangle$ of agent configurations $A_i = \langle i, bels(A_i), goals(A_i), plans(A_i)\rangle$ of individual agents, and the state of the shared environment χ. 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 $St \subseteq St_1 \times \cdots \times St_k \times St_E$. 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.

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 is a plan that specifies what i is going to do in every configuration, i.e., $s_i : St_i \to Act$ such that $bels(A_i) \models_{cwa} bprec_i(s_i(A_i))$. A *collective strategy* for a group of agents $A = \{a_1, \ldots, a_r\} \subseteq \mathbb{A}\mathrm{gt}$ 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*[5] 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 $\lambda^{act}[j]$. Function $out(q, s_A)$ returns the set of all paths that may occur when agents A execute strategy s_A from state q onward:

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

$out(q, s_A) = \{\lambda = q_0(i_0 : \alpha_0)q_1(i_1 : \alpha_1)q_2 \ldots \mid q_0 = q$ 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 2ATL* can be now given by the following clauses:

$M, q \models p$ iff $q_E \models_{cwa} p$;
$M, q \models \mathsf{Bel}_i\varphi_0$ iff $bels(\lambda^{st}[0]_i) \models_{cwa} \varphi_0$;
$M, q \models \mathsf{Goal}_i\varphi_0$ iff $goals(\lambda^{st}[0]_i) \models_{cwa} \varphi_0$;
$M, q \models \neg\varphi$ iff $M, q \not\models \varphi$;
$M, q \models \varphi_1 \wedge \varphi_2$ iff $M, q \models \varphi_1$ and $M, q \models \varphi_2$;
$M, q \models \langle\!\langle A \rangle\!\rangle\gamma$ 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 $0 \le j < k$.

Formula φ is *valid in program P* if it holds in every state q of the model of P. The formula is *valid for 2APL* if it is valid in every 2APL program. Formula φ is *satisfiable* if there is a program and a state q in its model so that φ holds in q.

We observe that the universal path quantifier A of CTL can be expressed with $\langle\!\langle\emptyset\rangle\!\rangle$. Note also 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.

5.3 Verification

Let P be a multi-agent program, M the model of the program, q a state in M, and φ a state formula of 2ATL. The model checking problem asks whether $M, q \models \varphi$. The following theorems are straightforward extensions of the results from [30, 23, 22]:

Theorem 1. *Model checking "vanilla" 2APL is $\mathbf{\Delta_2^P}$-complete with respect to the number of transitions in model M and the length of formula φ.*

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

Theorem 3. *Model checking 2APL* and "vanilla" 2ATL is $\mathbf{PSPACE}$-complete with respect to the size of program P and the length of φ.*

That is, the complexity of verification with 2ATL 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* with respect to any kind of concurrent programs (cf. [29]). Several techniques can be applied to reduce the complexity of models and facilitate model checking – most notably, *abstraction* [11] and *partial order reduction* [12]. None of the techniques have been applied to strategic logics yet; we plan to study this issue in our future work.

6 Conclusions

In this paper, we present an adaptation of the strategic logic ATL to reasoning about multi-agent 2APL programs. We carefully outline possible choices regarding the model of execution, specification language, and semantics of formulae. Then, we use a variant of alternating-time temporal logic with imperfect information and imperfect recall, and interpret its formulae over asynchronous labeled transition systems that arise from the operational semantics of 2APL. Initial complexity results for the model checking problem suggest that verification is feasible only for relatively small programs.

This is a preliminary study, and the solutions we use in this paper are not necessarily final. Several other modeling and semantic choices are worth exploring; in particular, modeling and reasoning about *fair* executions of programs should be studied, as well as execution strategies that impose fairness. We also plan to investigate techniques that can help to reduce the practical complexity of verification, especially abstraction and partial order reduction.

References

1. N. Alechina, M. Dastani, B. Logan, and J.-J. Ch. Meyer. A logic of agent programs. In *Proceedings of AAAI*, pages 795–800, 2007.
2. N. Alechina, B. Logan, M. Dastani, and J.-J. Ch. Meyer. Reasoning about agent execution strategies. In *Proceedings of AAMAS*, pages 1455–1458, 2008.
3. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time Temporal Logic. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 100–109. IEEE Computer Society Press, 1997.
4. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time Temporal Logic. *Journal of the ACM*, 49:672–713, 2002.
5. L. Astefanoaei, M. Dastani, F.S. de Boer, and J.-J. Ch. Meyer. A verification framework for normative multi-agent systems. In *Proceedings of PRIMA 2008*, volume 5357 of *LNCS*. Springer, 2008.
6. N. Belnap and M. Perloff. Seeing to it that: a canonical form for agentives. *Theoria*, 54:175–199, 1988.
7. M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multiagent systems. *Journal and Logic and Computation*, 8(3):401–423, 1998.
8. 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.
9. 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.
10. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of Logics of Programs Workshop*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, 1981.
11. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
12. E.M. 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.
13. P.R. Cohen and H.J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

14. M. Dastani. 2APL: a practical agent programming language. *International Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 16(3):214–248, 2008.
15. E.A. Emerson and J.Y. Halpern. ”sometimes” and ”not never” revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.
16. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.
17. D. Harel and D. Kozen. Process logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144–170, 1982.
18. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
19. J. G. Henriksen and P. S. Thiagarajan. Dynamic linear time temporal logic. *Annals of Pure and Applied Logic*, 96(1–3):187–207, 1999.
20. K. Hindriks. Modules as policy-based intentions: Modular agent programming in GOAL. In *Procedings of ProMAS'07*, volume 4908 of *LNCS*. Springer, 2008.
21. 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.
22. W. Jamroga and T. Ågotnes. Modular interpreted systems. In *Proceedings of AAMAS'07*, pages 892–899, 2007.
23. W. Jamroga and J. Dix. Model checking ATL$_{ir}$ is indeed Δ_2^P-complete. In *Proceedings of EUMAS'06*, 2006.
24. W. Jamroga and W. van der Hoek. Agents that know how to play. *Fundamenta Informaticae*, 63(2–3):185–219, 2004.
25. P. Novák and W. Jamroga. Code patterns for agent oriented programming. In *Proceedings of AAMAS'09*, pages 105–112, 2009.
26. M. Pauly. A modal logic for coalitional power in games. *Journal of Logic and Computation*, 12(1):149–166, 2002.
27. A. Pnueli. The temporal logic of programs. In *Proceedings of FOCS*, pages 46–57, 1977.
28. A.S. Rao and M.P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484, 1991.
29. Ph. Schnoebelen. The complexity of temporal model checking. In *Advances in Modal Logics, Proceedings of AiML 2002*. World Scientific, 2003.
30. P. Y. Schobbens. Alternating-time logic with imperfect recall. *Electronic Notes in Theoretical Computer Science*, 85(2), 2004.