Agent Composition Synthesis based on ATL

Giuseppe De Giacomo and Paolo Felli
Dipartimento di Informatica e Sistemistica
SAPIENZA - Università di Roma
Via Ariosto 25 - 00185 Roma, Italy
{degiacomo,felli}@dis.uniroma1.it

ABSTRACT

Agent composition is the problem of realizing a "virtual" agent by suitably directing a set of available "concrete", i.e., already implemented, agents. It is a synthesis problem, since its solution amounts to synthesizing a controller that suitably directs the available agents. Agent composition has its roots in certain forms of service composition advocated for SOA, and it has been recently actively studied by AI and Agents community. In this paper, we show that agent composition can be solved by ATL (Alternating-time Temporal Logic) model checking. This results is of interest for at least two contrasting reasons. First, from the point of view of agent composition, it gives access to some of the most modern model checking techniques and state of the art tools, such as MCMAS, that have been recently developed by the Agent community. Second, from the point of view of ATL verification tools, it gives a novel concrete problem to look at, which puts emphasis on actually synthesize winning policies (the controller) instead of just checking that they exist.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

General Terms

Theory, Verification, Algorithms

Keywords

Agent composition, synthesis, model checking, ATL

1. INTRODUCTION

Agent composition is the problem of realizing a "virtual" agent by suitably directing a set of available "concrete", i.e., already implemented, agents. It is a synthesis problem, whose solution amounts to synthesizing a controller that suitably directs the available agents.

Agent composition has its roots in certain forms of service composition advocated for SOA [20]. However agents provide a much more sophisticated context for the problem, and in the last years, the research on agent composition within the AI and Agents community has been quite fruitful and several composition techniques have been devised, based on reduction to PDL satisfiability [6, 5, 17], on forms of simulation or bisimulation [10, 18, 4, 2], on LTL (Linear time logic) synthesis [15, 14, 9, 12] and on direct techniques [19].

In this paper, we show that agent composition can be solved by ATL model checking. ATL (Alternating-time Temporal Logic) [1] is a logic whose interpretation structures are multi-player game structures where players can collaborate or confront each other so as to satisfy certain formulae. Technically, ATL is quite close to CTL, with which it shares excellent model checking techniques [3]. Differently from CTL, when an ATL formula is satisfied then it means that there exists a strategy, for the players specified in the formula, that fullils the temporal/dynamic requirements in the formula. ATL has been widely adopted by the Agents community since it allows for naturally specifying properties of societies of agents [21, 8]. The interest of the Agents community has led to active research on specific model checking tools for ATL, which by now are among the best model checkers for verification of temporal properties [7].

We show that indeed agent composition can be naturally expressed as checking a certain ATL formula over a specific game structure where the players are the virtual target agent, the concrete available agents, and a controller, whose actual controlling strategy has yet to be defined. The players corresponding to the target and to the available agents team up togheter against the controller. The controller tries to realize the target by looking, at each point in time, at the action chosen by the target agent, and by selecting accordingly who, among the available agents, actually performs the action. In doing this the controller has to cope with the choice of the action to perform by the target agent and the nondeterministic choice of the next state of the available agent that has been selected to perform the action. The ATL formula essentially requires that the controller avoids errors, where an error is produced whenever no available agents are able to actually perform the target agent's action currently requested. If the controller has a strategy to satisfy the ATL formula, then, from such strategy, a refined controller realizing the composition can be synthesized. In fact, we show that by ATL model checking we get much more than a single controller realizing a composition: we get a "controller generator" [18] i.e., an implicit representation of all possible controllers realizing a composition.

The results of this paper are of interest for at least two contrasting reasons. First, from the point of view of agent composition, it gives access to some of the most modern

Cite as: Agent Composition Synthesis based on ATL, Giuseppe De Giacomo, Paolo Felli, *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. 499-506

model checking techniques and tools, such as MCMAS, that have been recently developed by the Agent community. Second, from the point of view of ATL verification tools, it gives a novel concrete problem to look at, which puts emphasis on actually synthesize winning policies (the refined controller) instead of just checking that they exist, as usual in many contexts where ATL is used for agent verification.

The rest of the paper is organized as follows. In Section 2, we formally introduce the notion of agent composition. In Section 3, we give some background notions on ATL needed in the paper. In Section 4, we devise the encoding of agent composition as an ATL model checking problem, and, in Section 5, we show the soundness and completeness of the proposed technique, as well as its optimality from the computational complexity point of view. In Section 6, we discuss how to use a concrete model checker for ATL, namely MCMAS, to do the composition synthesis. In Section 7, we conclude the paper with a brief discussion on future work.

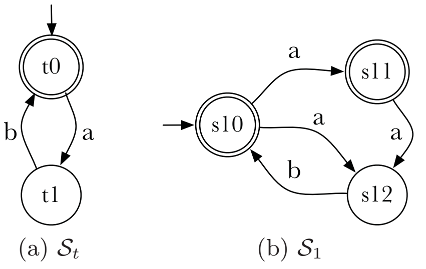
2. AGENT COMPOSITION

In this paper we address the *Agent Composition Problem* following the approach proposed in [19, 17, 18]. In such an approach, agents are characterized by their behaviour, modeled as a *transition system* (TS), which captures the agent executions, as well as the available choices that, at each point, the agent has available for continuing its execution. Given a *virtual target agent*, i.e., an agent of which we have the desired behavior but not its actual implementation, and a set of available concrete agents, i.e., a set of agents, each with its own behavior, that are indeed implemented, the composition's goal is to synthesize a *controller*, i.e., a suitable software module, capable of implementing the target agent by suitably controlling the available agents. Such a module realizes a target agent if and only if it's able, at every step, to delegate every action executable by the target to one of the available agent. Notice that, in doing this, the controller has to take into account not only local states of both the target and the available agents, but also their future evolutions, delegating actions to available agents so that all possible future target agent's actions can continue to be delegated. We call such a controller a *composition* of the available agent that realizes the target agent.

Formally, an *agent* is a transition system, i.e., a tuple $\mathcal{S} = \langle \mathcal{A}, S, s_0, \delta, F \rangle$ where:

- $\mathcal{A}$ is the finite set of actions;
- S is the finite set of states;
- s_0 is the initial state;
- $\delta \subseteq S \times \mathcal{A} \times S$ is the transition relation;
- $F \subseteq S$ is the set of final states.

We often write $s \xrightarrow{a} s'$ instead of $\langle s, a, s' \rangle \in \delta$. We assume that, in each state s, there is at least one action a that the agent can perform, i.e., there exists an s' such that $s \xrightarrow{a} s'$. The agent *can* (but does not *need* to) legally terminate whenever it is in a final state $s \in F$. Note that, in general, agents are *non-deterministic*: δ is defined as a transition relation; thus the state reached after performing action $a \in \mathcal{A}$ from state $s \in S$ cannot be foreseen. When the transition relation is in fact a partial function from $S \times \mathcal{A}$ to



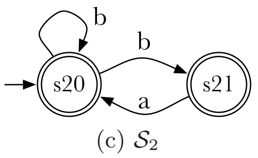


Figure 1: Target agent $\mathcal{S}_t$ and available agents $\mathcal{S}_1, \mathcal{S}_2$

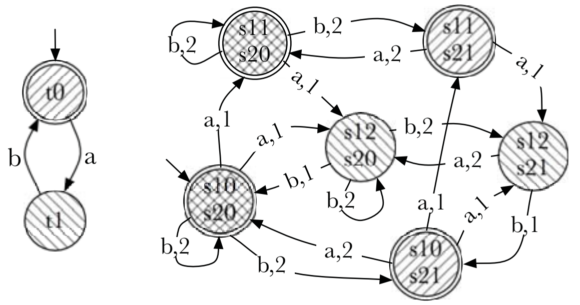


Figure 2: $\mathcal{S}_t$ simulated by $\mathcal{S}_1, \mathcal{S}_2$

S we say that the agent is *deterministic*. We say that non-deterministic agents are *partially (action) controllable* in the sense that when the agent is instructed to do an action, the actual resulting state is unpredictable by the controller. Conversely, we say that a deterministic agent is *fully (action) controllable*. We assume that the available agents are partially controllable while the target agent, i.e., the agent that we want to realize, is *fully controllable*.

Figure 1 shows the graphic representation of a target agent $\mathcal{S}_t$ and two available agents $\mathcal{S}_1$ and $\mathcal{S}_2$. Following a well-established convention, we graphically represent states as circles (nodes) and transitions as arrows (edges) labeled with actions. Final states are double-circled.

In [18] it has been shown that checking for the existence of an agent composition is equivalent to checking for the existence of a variant of the simulation relation [10] between the target agent and the available agents. Such a *(non-deterministic) simulation relation* can be defined as follows.

Given a target agent $\mathcal{S}_t$ and n available agents $\mathcal{S}_1, \ldots, \mathcal{S}_n$ with $\mathcal{S}_i = \langle \mathcal{A}, S_i, s_{i0}, \delta, F_i \rangle$ and $i = t, 1, \cdots, n$, a *simulation relation* of $\mathcal{S}_t$ by $\mathcal{S}_1, \ldots, \mathcal{S}_n$ is a relation $R \subseteq \mathcal{S}_t \times \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$ such that $\langle s_t, s_1, \ldots s_n \rangle \in R$ implies:

- if $s_t \in F_t$ then $s_i \in F_i$ for $i = 1, \cdots, n$;
- for each transition $s_t \xrightarrow{a} s'_t$ in $\mathcal{S}_t$ there exists an index $j \in \{1, \ldots, n\}$ such that the following holds:
  - there exists at least one transition $s_j \xrightarrow{a} s'_j$ in $\mathcal{S}_j$;
  - for *all* transitions $s_j \xrightarrow{a} s'_j$ in $\mathcal{S}_j$ we have that $\langle s'_t, s_1 \ldots, s'_j \ldots, s_n \rangle \in R$ (all agents but $\mathcal{S}_|$ remain still).

Let $\mathcal{S}_t$ be the target agent and $\mathcal{S}_1 \dots, \mathcal{S}_n$ be the available agents. A state $s_t \in S_t$ is *simulated by* a state $\langle s_1, \dots, s_n \rangle \in S_1 \times \dots \times S_n$ ($\langle s_1, \dots, s_n \rangle$ *simulates* s_t), denoted $s_t \preceq \langle s_1, \dots, s_n \rangle$, if and only if there exists a simulation relation R of $\mathcal{S}_t$ by $\mathcal{S}_1 \dots, \mathcal{S}_n$ such that $R(s_t, s_1, \dots, s_n)$. Extending this notion to the whole agents, we say that $\mathcal{S}_t$ is *simulated by* $\mathcal{S}_1 \dots, \mathcal{S}_n$ (or $\mathcal{S}_1 \dots, \mathcal{S}_n$ *simulate* $\mathcal{S}_t$) iff $s_{0t} \preceq \langle s_{01}, \dots, s_{0n} \rangle$, where s_{0t} and s_{0i}, with $i = 1, \dots, n$, are the initial states of the target agent and of the available agents, respectively. Figure 2 shows a graphical representation of the simulation relation R between target agent the available agents, where filling patterns (possibly overlapping) are used to denote *similar* states.

As shown in [18], we obtain the following fundamental result:

THEOREM 1. *[18] A composition of the available agents $\mathcal{S}_1, \dots, \mathcal{S}_n$ realizing the target agent $\mathcal{S}_t$ exists if and only if $\mathcal{S}_t$ is simulated by $\mathcal{S}_1, \dots, \mathcal{S}_n$.*

In other words, in order to checking for the existence of a composition it is sufficient to (i) compute the maximal simulation relation of $\mathcal{S}_t$ by $\mathcal{S}_1, \dots, \mathcal{S}_n$ and (ii) check whether $\langle s_{0t}, s_{01}, \dots, s_{0n} \rangle$ is in it.

Theorem 1 thus relates the notion of simulation relation to the one of agent composition showing, basically, that checking for the *existence* of an agent composition is equivalent to checking for the existence of a simulation relation between the target agent and the available agents. To actually synthesize a controller from the simulation we compute the so called *composition generator*, or *CG* for short. Intuitively, the *CG* is a program that returns, for each state the available agents may potentially reach while realizing a target history, and for each action the target agent may do in such a state, the set of all available agents able to perform the target agent's action, while guaranteeing that every future target agent's actions can still be fulfilled. The *CG* is directly obtained by the maximal simulation relation as follows:

Definition 1. (Composition Generator) Let $\mathcal{S}_t$ be a target agent and $\mathcal{S}_1, \dots, \mathcal{S}_n$ be n available agents, sharing the set of actions $\mathcal{A}$, such that $\mathcal{S}_t$ is simulated by $\mathcal{S}_1, \dots, \mathcal{S}_n$ and let $S_g = \{\langle s_t, s_1, \dots, s_n \rangle \mid s_t \prec \langle s_1, \dots, s_n \rangle\}$. The *Composition Generator (CG)* for $\mathcal{S}_t$ by $\mathcal{S}_1, \dots, \mathcal{S}_n$ is the function:

$\omega_g : S_g \times \mathcal{A} \to 2^{\{1,\dots,n\}}$ such that for $s_g = \langle s_t, s_1, \dots, s_n \rangle \in S_g$ and $a \in \mathcal{A}$

$$\omega_g(s_g, a) = \{i \mid \begin{array}{l} s_t \xrightarrow{a} s'_t \text{ is in } \mathcal{S}_t \text{ and} \\ s_i \xrightarrow{a} s'_i \text{ is in } \mathcal{S}_i \text{ and} \\ s'_t \preceq \langle s_1, \dots, s'_i, \dots, s_n \rangle \} \end{array}$$

CG is a function ω_g that given the states of the target and available agents, which are in simulation, and given an action, outputs the set of all available agents able to perform that action in their current state, while preserving the simulation. If there exists a composition of $\mathcal{S}_t$ by $\mathcal{S}_1, \dots, \mathcal{S}_n$, then the composition generator *CG generates* compositions, called *generated compositions*, by picking up one among the available agents returned by function ω_g, at each step of the (virtual) target agent execution, starting with all (target and available) agents in their respective initial state.

Next theorem guarantees that all compositions can be generated by the composition generator.

THEOREM 2. *[18] Let $\mathcal{S}_t$ and $\mathcal{S}_1, \dots, \mathcal{S}_n$ be as above. A controller P of $s_{01}, \dots, s_{0n}$ for $\mathcal{S}_t$ is a composition of $\mathcal{S}_1, \dots, \mathcal{S}_n$ realizing $\mathcal{S}_t$ if and only if it is a generated composition.*

3. ATL

Alternating-time Temporal Logic [1] is a logic that can predicate on moves of a game played by a set of players. For example, let Σ be the set of players and $A \subseteq \Sigma$, then the ATL formula $\langle\langle A \rangle\rangle \varphi$ asserts that there exists a *strategy* for players in A to satisfy the state predicate φ irrespective of how players in $\Sigma \backslash A$ evolve. The temporal operators are "$\Diamond$" (eventually), "$\Box$" (always), "$\bigcirc$" (next) and "$\mathcal{U}$" (until). The ATL formula $\langle\langle p1, p2 \rangle\rangle \Diamond \varphi$ captures the requirement "players $p1$ and $p2$ can cooperate to eventually make φ true". This means that there exists at a winning strategy that $p1$ and $p2$ can follow to force the game to reach a state where φ is true.

ATL formulae are constructed inductively as follows:

- p, for propositions $p \in \Pi$ are ATL formulae;
- $\neg\varphi$ and $\varphi_1 \vee \varphi_2$ where φ, φ_1 and φ_2 are ATL formulae, are ATL formulae;
- $\langle\langle A \rangle\rangle \bigcirc \varphi$ and $\langle\langle A \rangle\rangle \Box \varphi$ and $\langle\langle A \rangle\rangle \varphi_1 \mathcal{U} \varphi_2$, where $A \subseteq \Sigma$ is a set of players and φ, φ_1 and φ_2 are ATL formulae, are ATL formulae.

We also use the usual boolean abbreviations.

ATL formulae are interpreted over *concurrent game structures*: every state transition of a concurrent game structure results from a set of moves, one for each player. Formally, such a structure is a tuple $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ where:

- $k \geq 1$ is the number of players, each identified by an index number: $\Sigma = \{1, \dots, k\}$.
- Q is a finite, non-empty, set of states.
- Π is a finite, non-empty, set of boolean, observable, state propositions.
- $\pi : Q \to 2^{\Pi}$ is a labeling function which returns the set of propositions satisfied in each $q \in Q$.
- In each state $q \in Q$, each player $a \in \{1, \dots, k\}$ has $d_a(q) \geq 1$ available moves, identified with numbers $\{1, \dots, d_a(q)\}$. A *move vector* for q is a tuple $\langle j_1, \dots, j_k \rangle$ such that $1 \leq j_a \leq d_a(q)$ for each player a. We denote with $D(q)$ the set $\{1, \dots, d_1(q)\} \times \dots \times \{1, \dots, d_k(q)\}$ of move vectors for $q \in Q$.
- For each state $q \in Q$ and each move vector $\langle j_1, \dots, j_k \rangle \in D(q)$, a state $q' = \delta(q, j_1, \dots, j_k) \in Q$ results from state q if every player $i \in \{1, \dots, k\}$ chooses move j_i. δ is called transition function and q' is said to be a *successor* of q.

Once the notion of successor is given, we can provide a formal definition of winning strategy: given a game structure S as above, a *strategy* for player $a \in \Sigma$ is a function f_a that maps every non-empty finite state sequence $\lambda \in Q^+$ to one of its moves, i.e., a natural number such that if the last state of λ is q then $f_a(\lambda) \leq d_a(q)$.

A *computation* of S is an infinite sequence $\lambda = q_0, q_1, q_2 \dots$ of states such that for each $i \geq 0$, the state q_{i+1} is a successor of q_i. The strategy f_a determines, for every finite prefix λ of a computation, a move $f_a(\lambda)$ for player a. Hence, a strategy f_a induces a set of computations that player a can enforce. Given a state $q \in Q$, a set $A \subseteq \{1, \ldots, k\}$ of players, and a set $F_a = \{f_a \mid a \in A\}$ of strategies, one for each player in A, we define the *outcomes* of F_a from q to be the set $out(q, F_A)$ of *q-computations* that the players in A collectively can enforce when they follow the strategies in F_A. A computation $\lambda = q0, q1, q2, \ldots$ is then in $out(q, F_A)$ if $q_0 = q$ and for all positions $i > 0$ every player a follows the strategy f_a to reach the state q_{i+1}, that is, there is a move vector $\langle j_1, \ldots, j_k \rangle \in D(q_i)$ such that $j_a = f_a(\lambda[0, i])$ for all players $a \in A$, and $\delta(q_i, j_1, \ldots, j_k) = q_{i+1}$.

Now we can provide a formal definition of the satisfaction relation: we write $S, q \models \varphi$ to indicate that the state q satisfies formula φ with respect to game structure S. $\models$ is defined inductively as follows:

- $q \models p$, for propositions $p \in \Pi$, iff $p \in \pi(q)$.
- $q \models \neg\varphi$ iff $q \not\models \varphi$.
- $q \models \varphi_1 \vee \varphi_2$ iff $q \models \varphi_1$ or $q \models \varphi_2$.
- $q \models \langle\langle A \rangle\rangle \bigcirc \varphi$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in out(q, F_A)$, we have $\lambda[1] \models \varphi$.
- $q \models \langle\langle A \rangle\rangle \Box \varphi$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in out(q, F_A)$ and all positions $i \geq 0$, we have $\lambda[i] \models \varphi$.
- $q \models \langle\langle A \rangle\rangle (\varphi_1 \mathcal{U} \varphi_2)$ iff there exists a set F_A of strategies, one for each player in A, such that for all computations $\lambda \in out(q, F_A)$, there exists a position $i \geq 0$ such that $\lambda[i] \models \varphi_2$ and for all positions $0 \leq j < i$, we have $\lambda[j] \models \varphi_1$.

As for operator "$\Diamond$" (eventually), we observe that $\langle\langle A \rangle\rangle \Diamond \varphi$ is equivalent to $\langle\langle A \rangle\rangle (true \mathcal{U} \varphi)$.

Concerning computational complexity, the cost of ATL model-checking is linear in the size of the game structure, as for CTL, a very well-known temporal logic used in model checking[3], of which ATL is an extension.

4. AGENT COMPOSITION VIA ATL

Now we look at how to use ATL for synthesizing compositions. To do so we introduce a concurrent game structure for the agent composition problem, reducing the search for possible compositions to the search for *winning strategies* in the multi-player game played over it. Given a target agent $\mathcal{S}_t$ and n available agents $\mathcal{S}_1, \ldots, \mathcal{S}_n$ with $\mathcal{S}_i = \langle \mathcal{A}, S_i, s0_i, \delta_i, F_i \rangle$ with $i = t, 1, \ldots n$, we define a game structure NGS for our problem as follows.

We start by slightly modifying the available agents $\mathcal{S}_i$ ($i = 1, \ldots, n$) by adding a new state err_i, disconnected, through δ_i, to the other states, and such that $err_i \notin F_i$.

We also define two convenient notations:

- $Act_i(s)$ that denotes the set of actions available to the agent i ($i = t, 1, \ldots, n$) in its local state s, i.e., $Act_i(s) = \{a \in \mathcal{A} \mid < s, a, s' > \in \delta_i \text{ for some } s'\}$.
- $Succ_i(s, a)$ that denotes the set of possible successor states for player i ($i = t, 1, \ldots, n$) when it performs action a from its local state s, i.e., $Succ_i(s, a) = \{s' \in S_i \mid < s, a, s' > \in \delta_i\}$.

The game structure $NGS = \langle k, Q, \Pi, \pi, d, \delta \rangle$ is defined as follows.

Players.

The set of players Σ is formed by one player for each available agent, one player for the target agent, and one player for the controller. Each player is identified by an integer $\Sigma = \{1, \ldots, k\}$.

- $i \in \{1 \ldots n\}$ for the available agents ($n = k - 2$)
- $t = k - 1$ is the target virtual agent
- k is the controller

Game structure states.

The states of the game structure are characterized by the following finite range functions:

- $state_i$: returns the current state of the agent i ($i = t, 1, \ldots, n$); it ranges over $s \in S_i$.
- sch : returns the scheduled available agent, i.e., the agent that performed the last action; it ranges over $i \in \{1, \ldots, n\}$.
- act_t : returns the action requested by the target, it ranges over $a \in \mathcal{A}$.
- $final_i$: returns whether the current state $state_i$ of agent i is final or not ($i = t, 1, \ldots, n$); it ranges over booleans.

Q is the set of states obtained by assigning a value to each of these functions, and Π is the set of propositions of the form $(f = v)$ corresponding to assert that function f has value v. Notice that we can use directly finite range functions, without fixing any specific encoding for the technical development that follows.

The function π, given a state q of the game structure returns the values for the various functions. For simplicity, we will use the notation $state_i(q) = s$ instead of $(state_i = s) \in \pi(q)$.

Initial states.

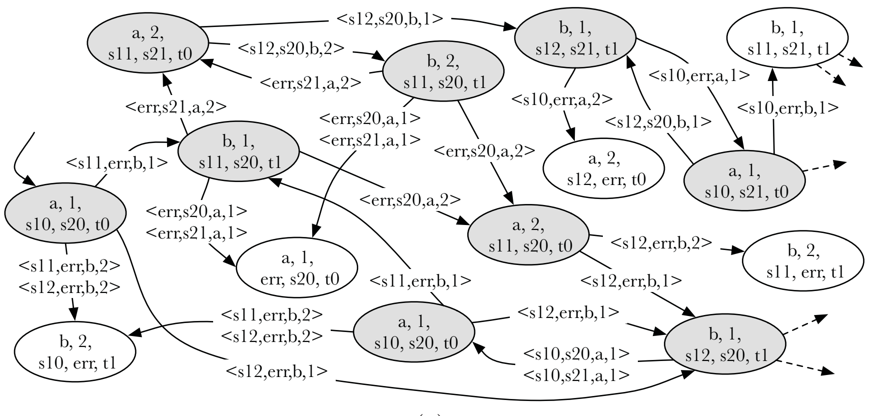
The initial states Q_0 of the game structure are those q_0 such that:

- every agent is in its local initial state, $state_i(q_0) = s_{0i}$ and $final_i(q_0) = true$ iff $s_{0i} \in F_i$ ($i = t, 1, \ldots, n$),
- $act_t(q_0) = a$ for some action $a \in Act(s_{0t})$, and
- $sch(q_0) = 1$ (this is a dummy value, which will be not used in any way during the game).

Players' moves.

The moves that the player i ($i = 1, \ldots, n$), representing the available agent $\mathcal{S}_i$, can perform in a state q are:

$$Moves_i(q) = \begin{cases} \{s' \mid s' \in Succ_i(state_i(q), act_t(q))\} \\ \qquad \text{if } Succ_i(state_i(q), act_t(q)) \neq \emptyset \\ \{err_i\} \qquad \text{otherwise.} \end{cases}$$



(a)

state$_1$	state$_2$	state$_t$	act	
s10	s20	t0	a	{1}
s10	s20	t1	b	{2}
s10	s21	t0	a	{2}
s11	s20	t0	a	{1}
s11	s20	t1	b	{2}
s11	s21	t0	a	{1,2}
s12	s20	t1	b	{1}
s12	s21	t1	b	{1}

(b)

Figure 3: (a) A fragment of a game structure and (b) the corresponding ω_{ACG}

The moves that the player k, representing the controller, can do in a state q are:

$$Moves_k(q) = \{1, \dots, n\}.$$

The moves that the player t, representing the target agent $\mathcal{S}_t$, can perform in a state q are (with a little abuse of notation, and recalling that the target agent is deterministic):

$$Moves_t(q) = Act_t(Succ(state_t(q), act_t(q))).$$

Notice that the player t chooses in the *current* turn the action that will be executed *next*.

The number of moves is $d_i(q) = |Moves_i(q)|$ and, wlog, we can associate some enumeration of the elements in $Moves_i(q)$.

Game transitions.

The game transition function δ is defined as follows: $\delta(q, j_1, \dots, j_k)$ is the game structure state q' such that:

- $sch(q') = j_k$
- $state_w(q') = j_w$ if $j_k = w$
- $state_i(q') = state_i(q)\ \forall i \neq w$
- $state_t(q') = s_t$, where $\{s_t\} = Succ(state_t(q), act_t(q))$
- $act_t(q') = j_t$
- $final_i(q') = true$ iff $state_i(q') \in F_i$.

Figure 3(a) shows a fragment of the game structure NGS for the example in Figure 1. Nodes represent states of the game and edges represent game transitions labelled with move vectors (for simplicity, states where one of the agents is in err are left as sink nodes).

ATL formula to check for composition.

Checking the existence of a composition is reduced to checking the ATL formula φ, over the game structure NGS, defined as follows:

$$\begin{aligned}\varphi \ = \ & \langle\langle k \rangle\rangle \Box(\\ & \quad \wedge_{i=1,\dots,n}(state_i \neq err_i) \wedge \\ & \quad (final_t \ \rightarrow \ (\wedge_{i=1,\dots,n} final_i = true)) \\ &)\end{aligned}$$

5. RESULTS

Given a target agent $\mathcal{S}_t$ and n available agents $\mathcal{S}_1, \dots, \mathcal{S}_n$, let $NGS = \langle k, Q, \Pi, \pi, d, \delta \rangle$ be the game structure and φ the ATL formula defined above. The set of winning states of the games is:

$$[\varphi]_{NGS} = \{q \in Q \mid q \models \varphi\}$$

Referring to Figure 3(a) grey states are those in $[\varphi]_{NGS}$.

From $[\varphi]_{NGS}$ we can build an ATL Composition Generator ACG for the composition of $\mathcal{S}_1, \dots, \mathcal{S}_n$ for $\mathcal{S}_t$ exploiting the set $[\varphi]_{NGS}$.

Definition 2. (ATL Composition Generator) Let NGS and φ be as above. We define the ATL Composition Generator ACG as a tuple $ACG = \langle \mathcal{A}, \{1, \dots, n\}, S_{NGS}, S^0_{NGS}, \omega_{ACG}, \delta_{ACG} \rangle$ where:

- $\mathcal{A}$ is the set of actions, and $\{1, \dots, n\}$ is the set of players representing the available agents, as in NGS;
- $S_{NGS} = \{\langle state_t(q), state_1(q), \dots, state_n(q) \rangle \mid q \in [\varphi]_{NGS}\}$;
- $S^0_{NGS} = \{\langle state_t(q_0), state_1(q_0), \dots, state_n(q_0) \rangle \mid q_0 \in Q_o \cap S_{NGS}\}$;
- $\delta_{ACG} : S_{NGS} \times \mathcal{A} \times \{1, \dots, n\} \rightarrow S_{NGS}$ is the transition function, defined as follows: $\langle s'_t, s'_1, \dots, s'_n \rangle \in \delta_{ACG}(\langle s_t, s_1, \dots, s_n \rangle, a, w)$ iff there exists $q \in [\varphi]_{NGS}$ with $s_i = state_i(q)$ for $i = t, 1, \dots, n$, $a = act_t(q)$, $s'_t \in Succ_t(s_t, a)$ such that *for each* $q' = \delta(q, s'_1, \cdots, s'_n, a', w)$, with $sch(q') = w$, $s'_w \in Succ_w(s_w, a)$, $s'_i = s_i$ for $i \neq w$, and $a' \in Act_t(q)$, we have $q' \in [\varphi]_{NGS}$.
- $\omega_{ACG} : S_{NGS} \times \mathcal{A} \rightarrow 2^{\{1,\dots,n\}}$ is the agent selection function: $\omega_{ACG}(\langle s_t, s_1, \dots, s_n \rangle, a) = \{i \mid \exists \langle s'_t, s'_1, \dots, s'_n \rangle$ with $\langle s'_t, s'_1, \dots, s'_n \rangle \in \delta_{ACG}(\langle s_t, s_1, \dots, s_n \rangle, a, i)\}$.

Figure 3(b) shows the agent selection function ω_{ACG} of the ATL Composition Generator for the game structure of Figure 3(a). Next theorem states the soundness and completeness of the method based on the construction of ACG for computing agent compositions.

Theorem 3. *Let $\mathcal{S}_t$ be a target agent and $\mathcal{S}_1, \ldots, \mathcal{S}_n$ n available agents. Let $ACG = \langle \mathcal{A}, \{1, \ldots, n\}, S_{ACG}, S^0_{ACG}, \omega_{ACG}, \delta_{ACG} \rangle$ and ω_g be, respectively, the ATL Composition Generator and the Composition Generator for $\mathcal{S}_t$ by $\mathcal{S}_1, \ldots, \mathcal{S}_n$. Then*

1. *$\langle s_t, s_1, \ldots, s_n \rangle \in S_{ACG}$ iff $s_t \preceq \langle s_1, \ldots, s_n \rangle$ and*
2. *for all $s_t, s_1, \ldots, s_n$ such that $s_t \preceq \langle s_1, \ldots, s_n \rangle$ and for all $a \in \mathcal{A}$, we have that*

$$\omega_{ACG}(\langle s_t, s_1, \ldots, s_n \rangle, a) = \omega_g(\langle s_t, s_1, \ldots, s_n \rangle, a)$$

Proof. We focus on (1) since (2) is a direct consequence of 1 and of the definition ω_{ACG}. ACG's correctness is basically proven showing that the set S in ACG is a simulation relation (i.e., it satisfies the constraints (i) and (ii) in the definition of simulation relation), and it is hence contained in $\preceq$ which is the largest one.

As for completeness, we show that there exists no generated composition P for $\mathcal{S}_t$ and $\mathcal{S}_1, \ldots, \mathcal{S}_n$ which cannot be generated by ω_{ACG}. Toward contradiction let us assume that one such P exists. Then there exists a history of the system coherent with P, such that, considering the definition of ACG either (a) the requested action can't be performed in target's current state s_t, (b) target's current state s_t is final but at least one of the current states of the s_i $(i = 1, \ldots, n)$ available agents is not, or (c) no available agent is able to perform the requested action in its own current state s_i $(i = 1, \ldots, n)$, that is if all successor game states reached after performing it are error states. But (a) cannot happen by construction of ACG being the history coherent with P, and if either of (b) and (c) happens we get that $s_t \not\preceq \langle s_1, \ldots, s_n \rangle$ contradicting the assumption that P is a generated composition. $\Box$

Analogously of what done for the composition generator in Section 2, we can define the notion of *ACG generated compositions*: i.e., the compositions obtained by picking up one among the available agents returned by function ω_{ACG}, at each step of the (virtual) target agent execution starting with all agents (target and available in their initial state). Then, as a direct consequence of Theorem 3 and the results of [18], we have that:

Theorem 4. *Let $\mathcal{S}_t$ be a target agent and $\mathcal{S}_1, \ldots, \mathcal{S}_n$ n available agents. Then (i) if $[\varphi]_{NGS} \neq \emptyset$ then every controller generated by ACG is a composition of target agent $\mathcal{S}_t$ by $\mathcal{S}_1, \ldots, \mathcal{S}_n$ and (ii) if such composition does exist, then $[\varphi]_{NGS} \neq \emptyset$ and every controller that is a composition of the target agent $\mathcal{S}_t$ by $\mathcal{S}_1, \ldots, \mathcal{S}_n$ can be generated by the ATL Composition Generator ACG.*

By recalling that model checking ATL formulas is linear in the size of the game structure, analyzing the construction above we have:

Theorem 5. *Computing ATL composition generator (ACG) is polynomial in the number of states of the target and available agents and exponential in the number of available agents.*

Proof. The results follows by the construction of the game structure NGS above and from the fact that model checking ATL formula over game structure can be done in polynomial time. $\Box$

From Theorem 4 and the EXPTIME-hardness of result in [11], we get a new proof of the complexity characterization of the agent composition problem [18].

Theorem 6. *[18] Computing agent composition is EXPTIME-complete.*

6. IMPLEMENTATION

In this section we show how to use the ATL model checker MCMAS [7] to solve agent composition via ATL model checking. In particular, following the definition of game structure NGS, we show how to encode instances of the agent composition problem in ISPL (Interpreted Systems Programming Language) which is the input formalism for MCMAS. For readability, we show here a basic encoding, according to the definition of NGS; some refinement will be discussed at the end of the section.

ISPL distinguishes between two kinds of agents: *ISPL standard agents* and one *ISPL Environment*. In brief, both ISPL standard agents and the ISPL Environment are characterized by (1) a set of local states, which are private with the exception of Environment's variables declared as `Obsvars`; (2) a set of actions, one of which is choosen by the ISPL agent in every state; (3) a rule describing which action can be performed by the ISPL agent in each local state (`Protocol`); and (4) a function describing how the local state evolve (`Evolution`).

We encode both the available agents and the target agent of our problem as ISPL standard agents, while we encode the controller in the Environment. Each ISPL standard agent features a variable `state`, holding the current state of the corresponding agent, while the ISPL Environment has two variables: `sch` and `act`, which correspond to propositions sch and act_t in Π, i.e., respectively, the available agent chosen by the controller to perform the requested target agent's action, and the target agent's action itself. The special value `start` is introduced for technical convenience: we need to "generate" a state for each possible action the target agent may request at the beginning of the game. All variables have enumeration type, ranging over the set of values they can assume according to the definition of NGS.

We illustrate the ISPL encoding of our running example. Consider the same available and target agents as in Figure 1. The code for the ISPL Environment `Environment`:

```
Semantics = SA;

Agent Environment
   Obsvars:
      sch : {S1,S2,start};
      act : {a,b,start};
   end Obsvars
   Actions = {S1,S2,start};
   Protocol:
      act=start : {start};
      Other : {S1,S2};
   end Protocol
   Evolution:
      sch=S1 if Action=S1;
      sch=S2 if Action=S2;
      act=a if T.Action=a;
      act=b if T.Action=b;
   end Evolution
end Agent
```

Notice that the values of `sch` are unconstrained; they depend on the action chosen by the environment, which chooses them so as to satisfy the ATL formula of interest. Instead, `act` stores the action that the target agent has chosen to do next. The statement `Semantics = SA` specifies that only one assignment is allowed in each evolution line. This implies that evolution items are partitioned into groups such that two items belong to the same group if and only if they update the same variable and that they are not mutually excluded as long as they belong to different groups.

Next we show the encoding as ISPL standard agents `S1` and `S2` for the available agents $S1$ and $S2$.

```
Agent S1
 Vars:
  state : {s10,s11,s12,err};
 end Vars
 Actions = {s10,s11,s12,err};
 Protocol:
  state=s10 and Environment.act=a : {s11,s12};
  state=s11 and Environment.act=a : {s12};
  state=s12 and Environment.act=b : {s10};
  Other : {err};
 end Protocol
 Evolution:
  state=err if Action=err and Environment.Action=S1;
  state=s10 if Action=s10 and Environment.Action=S1;
  state=s11 if Action=s11 and Environment.Action=S1;
  state=s12 if Action=s12 and Environment.Action=S1;
 end Evolution
end Agent
```

```
Agent S2
 Vars:
  state : {s20,s21,err};
 end Vars
 Actions = {s20,s21,err};
 Protocol:
  state=s20 and Environment.act=b : {s20,s21};
  state=s21 and Environment.act=a : {s20};
  Other : {err};
 end Protocol
 Evolution:
  state=err if Action=err and Environment.Action=S2;
  state=s20 if Action=s20 and Environment.Action=S2;
  state=s21 if Action=s21 and Environment.Action=S2;
 end Evolution
end Agent
```

Each ISPL standard agent for the available agents *reads* variable `Environment.act` which has been chosen in the previous game round and chooses a next state to go to among those reachable through that action. If such an action is not available to the agent in its current state, then `err` is chosen. If the ISPL standard agent is the one chosen by the controller, then by reaching such an error state, it falsifies the ATL formula.

Finally, we show the encoding as a ISPL standard agent `T` of the target agent T .

```
Agent T
   Vars:
      state : {t0,t1};
   end Vars
   Actions = {a,b};
   Protocol:
      Environment.act=start: {a};
      state=t0 and Environment.act=a : {b};
      state=t1 and Environment.act=b : {a};
   end Protocol
   Evolution:
      state=t1 if state=t0 and Environment.act=a;
      state=t0 if state=t1 and Environment.act=b;
   end Evolution
end Agent
```

The ISPL standard agent `T` reads the current action `Environment.act` in the ISPL Environment `Environment`, which stores its own previous choice, and virtually makes *the* corresponding transition (remember that the target agent is deterministic) getting to the new state. Then, it selects its next action among those available in its next state. Consider for example the first `Evolution` statement: `state=t1 if state=t0 and Environment.act=a`. Such a can be read as follows: *"if current state is `t0` and the (last) action requested is `a`, then request an action chosen among those available at state `t1`, namely the set `{b}` in this case"*. Note that, considering the definition of `Environment`, the ISPL standard agent `T` chooses the action to be stored in `Environment.act` at the *next* turn of the game.

The ISPL code is completed as follows.

```
Evaluation
  Error if S1.state=err or S2.state=err;
  S1Final if S1.state=s10 or S1.state=s11;
  S2Final if S2.state=s20 or S2.state=s21;
  TFinal if T.state=t0;
end Evaluation

InitStates
  S1.state=s10 and S2.state=s20 and T.state=t0 and
  Environment.act=start and Environment.sch=start;
end InitStates

Groups
  Controller = {Environment};
end Groups

Formulae
  <Controller> G (
    !Error and (TFinal -> (S1Final and S2Final))
  );
end Formulae
```

where we define some computed propositions for convenience (`Evaluation`), the initial state of the game (`InitState`), the group of agents appearing in the ATL formula (`Groups`), and the ATL formula itself (`Formulae`). All of these part directly correspond to what described in Section 4: in particular the ATL formula requires that all ISPL standard agents for available agents have to be in a final state if the one for the target does, and none of the ISPL standard agent for available agents can be in error state (since this can only be reached whenever the scheduled available agent cannot actually replicate requested action).

Standard MCMAS checks if the ATL formula φ is satisfied in the specified game structure NGS. However, we used an available prototype of MCMAS that can actually return a convenient data structure with the set of all states of the game structure that satisfy the ATL formula, namely the set $[\varphi]_{NGS}$, and the transitions among them. Using such a data structure we wrote a simple Java program that actually computes ω_{ACG} of Definition 2, thus obtaining a practical way to generate all compositions.

The whole approach is quite effective in practice. In particular, we have run some experimental comparisons with direct implementations of the simulation approach proposed in [18], and our MCMAS based system is generally two orders of magnitude faster. We also compare it with implementations based on TLV [14] that are tailored for the agent composition problem [12], and the results of the two systems are similar, even if we used a completely standard MCMAS implementation and prototypical additional components.

We close the section by observing that the ISPL encoding shown here, which directly reflects the theoretical construction done above, could be easily refined (though possibly at expenses of clarity) with at least two major improvements. First, we can massively reduce the number of error states from the resulting structure, giving ISPL Environment both a copy of available agents' states and a protocol function which only schedules ISPL standard agents that are actually able to perform the current target action, according to their own protocols. If none of the ISPL standard agents is able to fullfill this condition, then an error *action* is selected, and the successor state is flagged with a boolean variable set to true in the `Environment`. Local error states are no more needed and the `Error` definition in the `Evaluation` section needs to be changed accordingly. Second, the system can be forced to "loop" after such an error condition is reached, e.g. forcing all ISPL agents to select an error action, thus avoiding to generate further (error) states.

7. CONCLUSIONS

In this paper we have shown that agent composition can be naturally and effectively solved by ATL model checking. This gives effective techniques for agent composition based on current ATL model checkers.

The connection between agent composition and ATL and more generally the work on ATL-based agent verification can be quite fruitful in the future. First of all, such a work can stimulate the ATL community on focusing more on the problem of actually extracting the strategies to lead to the satisfaction of ATL formulae, moving from ATL-based verification to ATL-based synthesis. Then several advancements in the recent work on ATL within the Agent community can be of great interest for agent composition. For example, the recent work on using ATL together with forms of epistemic logics to capture the different knowledge of the various agents could give effective techniques to deal with partial observability of the behaviors of the available agents in the agent composition. Currently known techniques are mostly based on a belief-state construction [16, 13, 4] and have mostly resisted effective implementations. We plan to look into this issue in the future.

Acknowledgments

We would like to thank Alessio Lomuscio, Hongyang Qu and Franco Raimondi for the development of some experimental components of MCMAS that return full witnesses of ATL formulae, which have proven to be essential for our work. We would like also to thank Riccardo De Masellis and Fabio Patrizi for discussions and insights, and the anonymous reviewers for their valuable comments.

8. REFERENCES

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

[2] P. Balbiani, F. Cheikh, and G. Feuillade. Algorithms and complexity of automata synthesis by asynhcronous orchestration with applications to web services composition. *Electronic Notes in Theoretical Computer Science*, 229(3):3–18, July 2009.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. The MIT Press, Cambridge, MA, USA, 1999.

[4] G. De Giacomo, R. De Masellis, and F. Patrizi. Composition of partially observable services exporting their behaviour. In *ICAPS*, 2009.

[5] G. De Giacomo and S. Sardiña. Automatic synthesis of new behaviors from a library of available behaviors. In *IJCAI*, pages 1866–1871, 2007.

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

[7] A. Lomuscio, H. Qu, and F. Raimondi. Mcmas: A model checker for the verification of multi-agent systems. In *CAV*, pages 682–688, 2009.

[8] A. Lomuscio and F. Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *AAMAS*, pages 161–168, 2006.

[9] Y. Lustig and M. Y. Vardi. Synthesis from component libraries. In *FOSSACS*, pages 395–409, 2009.

[10] R. Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489, 1971.

[11] A. Muscholl and I. Walukiewicz. A lower bound on web services composition. *Logical Methods in Computer Science*, 4(2), 2008.

[12] F. Patrizi. *Simulation-based Techniques for Automated Service Composition*. PhD thesis, DIS, Sapienza Univ. Roma, 2009.

[13] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI*, pages 1252–1259, 2005.

[14] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.

[15] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.

[16] J. Rintanen. Complexity of planning with partial observability. In *ICAPS*, pages 345–354, 2004.

[17] S. Sardiña, F. Patrizi, and G. De Giacomo. Automatic synthesis of a global behavior from multiple distributed behaviors. In *AAAI*, 2007.

[18] S. Sardiña, F. Patrizi, and G. De Giacomo. Behavior composition in the presence of failure. In *KR*, pages 640–650, 2008.

[19] T. Ströder and M. Pagnucco. Realising deterministic behavior from multiple non-deterministic behaviors. In *IJCAI*, 2009.

[20] J. Su, editor. *IEEE Data Engineering Bulletin: Special Issue on Semantic Web Services*, volume 31:2, 2008.

[21] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2nd edition, 2009.