Composition of stateful deterministic services in ATL

Paolo Felli

paolo.felli@gmail.com

Matteo Vita

matteo.vita@gmail.com

Dipartimento di Informatica e Sistemistica

Sapienza Università di Roma

Outline

Our purpose is to show how a Service Composition Problem instance can be encoded into a concurrent game structure, and how searching for a composition is equivalent to searching for a winning strategy for a corresponding multi-player game.

We'll make use of Alternating-time Temporal Logic (ATL).

Web Services

Web Services are self-describing computational elements representing software modules capable of performing actions. These modules are intended to interact with a client and their interactions follow a given behavior.

Services can be characterized in different ways. In our approach services are described by their conversational behaviour, modeled as **transition systems** that capture the possible conversations a service can have with its clients.

Transition systems

Transition System TS = <A,S,s0,δ,F> is defined as follows:

- A is the finite alphabet of actions
- S is the finite set of states
- s0 is the initial state
- δ is the transition relation $\delta \subseteq S \times A \times S$
- F is the set of final states

Service composition

Our goal is, given a target service specifying a desired interaction with the client, to synthesize an **orchestrator** capable of realizing it exploiting a set of available services.

Notably the target service itself is represented by a transition system TS_t but it is not one of the available services.

Orchestrator

An orchestrator is, basically, a function which selects an available service for executing the action requested, maintaining with the client the same (infinite) interaction that it would have with the target service.

Assumptions

1. The orchestrator has full observability on the available services and it can keep track at runtime of their current state.

2. All services are stateful and deterministic (fully controllable).

Alternating-time Temporal Logic

We briefly discuss Alternating-time Temporal Logic (ATL). In particular:

- Why to use ATL temporal logic
- ATL syntax and semantics
- The Composition Problem in ATL

Open systems

A closed system is a system whose behavior is completely determined by its own state.

However we need to model reactive systems, in which each component behaves as an open system that interacts with its environment and whose behavior is determined by the state of the system as well as the state of the environment.

Satisfaction Problem

Alternation can be considered as a natural generalization of existential and universal branching.

- **Universal satisfaction** (LTL):
do all computations satisfy a property?

- **Existential satisfaction** (CTL):
does some computation satisfy a property?

- **Alternating satisfaction** (ATL):
can the system resolve its internal choises so that the satisfaction of a property is guaranteed no matter how the environment resolves the external choices?

Alternating satisfaction

ATL offers selective quantification over paths seen as possible outcomes of a game between a system and the environment.

→ alternating satisfaction can be viewed as a winning condition in this game.

Concurrent Game Structure

While modeling language for open system use a variety of different communication mechanism, they can be given a common semantics in terms of **Concurrent Game Structure** (CGS) in which:

- A concurrent game is played on a state space
- At each step each player chooses a move
- A transition is determined by a combination of choices.

CGS types

There are three types of game structures:

Turn-based synchronous : at each step, only one player has a choice of moves, and this player is determined by the current state.

Moore synchronous : at each step all players proceed simultaneously choosing their next state independently of the moves chosen by the others.

Turn-based asynchronous : at each step, only one player has a choice of moves, and that player is chosen by a fair scheduler.

Multi-player games

In order to capture composition of open systems we consider, instead of 2-player game, the more general setting of multi-player game, with a finite **set of players** that represent all the different components involved.

Multi-player games

Consider a game between a **protagonist** and an **antagonist**. Consider a set $A \subseteq \Sigma$ of players, a set L of computations, and a state q of the system.

Starting from state q, at each step the protagonist moves players in A (he chooses a move for each of them) while the antagonist resolves the remaining choices.

If the infinite computation resulting from this game belongs to set L, then the antagonist wins; otherwise he loses.

Winning strategies

If the protagonist can actually win the game, then exist a winning strategy that the players in A can follow to force a computation in L, **irrespective** of how the players in Σ\A choose their moves.

We say that the ATL formula ≪A≫L is satisfied in the state q.

Hence ≪A≫ is a generalization of CTL path quantifiers:

- the existential path quantifier ∃ is equivalent to ≪Σ≫
- the universal ∀ corresponds to ≪≫.

ATL syntax

The temporal logic ATL is defined with respect to a finite set Π of propositions and afinite set Σ = {1...k} of players. An ATL formula is one of the following:

1. **p**, for propositions p ∈ Π.
2. **¬φ** or **φ1 ∨ φ2**, where φ, φ1 and φ2 are ATL formulas.
3. **≪A≫○φ**, **≪A≫□φ** or **≪A≫φ1Uφ2**, where A ⊆ Σ is a set of players and φ, φ1 and φ2 are ATL formulas.

The operator ≪·≫ is a path quantifier, and ○ (“next”), □(“always”), and U (“until”) are temporal operators.

We write ≪A≫◇φ for ≪A≫trueUφ.

ATL semantics

We write **S,q|=ψ** to indicate that the state q satisfies the formula ψ in the structure S. The satisfaction relation |= is inductively defined, for all states q, as follows:

- **q |= p**, for propositions p ∈ Π, iff p ∈ π(q).
- **q |= ¬φ** iff q | ≠φ.
- **q |= φ1 ∨ φ2** iff q |= φ1 or q |= φ2.
- **q |= ≪A≫○φ** iff there exists a set F_A of strategies, one for each player in A, such that for all computations λ∈ out(q, F_A), we have λ[1]|= φ.
- **q |= ≪A≫□φ** iff there exists a set F_A of strategies, one for each player in A, such that for all computations λ∈ out(q, F_A) and all positions i ≥ 0, we have λ[i] |= φ.
- **q |= ≪A≫φ1Uφ2** iff there exists a set FA of strategies, one for each player in A, such that for all computations λ ∈ out(q, F_A), there exists a position i ≥ 0 such that λ[i] |= φ2 and for all positions 0 ≤ j<i , we have λ[j] |= φ1

ATL semantics

The dual form for ≪A≫ is [[A]] : while ≪A≫φ intuitively means that the players in A can cooperate to make φ true (can enforce φ), [[A]]φ means that the players in A cannot cooperate to make φ false.

Hence ≪A≫ can be viewed as a path quantifier, parameterized with the set A of players, which ranges over all computations that the players in A can force the game into, irrespective of how the players Σ\A proceed.

Composition problem in ATL

As stated before, we intend to reduce the search for a composition to the search for winning strategies for the corresponding multi-player game over a concurrent game structure.

We chosen the **Turn-based Asynchronous** Game Structure.

Concurrent Game Structure

A Concurrent Game Structure is a tuple
$S=<k,Q,\Pi,\pi,d,\delta>$

- A natural number $k \geq 1$ of players.
- A finite set Q of states.
- A finite set Π of observable propositions.
- $\forall q \in Q$ a set $\pi(q) \in \Pi$ of propositions true at q

Concurrent Game Structure

- For each player $a \in \{1..k\}$, a natural number $d_a(q) \geq 1$ of moves available at state q for player a. We identify the moves of a player a at state q with the number $1...d_a(q)$.

- $\forall q \in Q$ and for each move vector $\langle j_1...j_k \rangle \in D(q)$, a state $\delta(q,j_1...j_k) \in Q$ that results from state q if every player $a \in \{1..k\}$ chooses move ja.

Turn-based Asynchronous GS

In a turn-based asynchronous game structure, one player is designed to represent a scheduler.

If the set of player is {1,...,k}, we assume that the scheduler is always player k.

In every state, the scheduler select one of the other k-1 players. Scheduled player completely determines the next state.

We say that a player $a \in \{1..k\}$ is *scheduled* whenever player k chooses move a.

Turn-based Asynchronous GS

Formally, a game structure $S = < k,Q,\Pi,\pi,d,\delta >$ is a turn-based asynchronous game structure if $k \geq 2$ and for every state $q \in Q$ the following two condition are satisfied:

- $d_k(q)=k-1$
- For all move vectors $<j_1 \ldots j_k>,<j'_1 \ldots j'_k> \in D(q)$

  if $j_k=j'_k$ and $j_a=j'_a$ for $a=j_k$

  then $\delta(q,j_1 \ldots j_k)=\delta(q,j'_1 \ldots j'_k)$

Composition Problem in ATL

We defined a CGS for the composition problem starting from the definition of Turn-based Asynchronous game structure.

$$GS = < k,Q,\Pi,\pi,d,\delta,\omega >$$

Composition Problem in ATL

We define the global alphabet of operation

$$A = A_1 \cup \ldots \cup A_{n+1}$$

For each one of the n available services we add an error state *s_err* to the set S_i

Composition Problem in ATL

All services partecipate to the game. Each of the k players is identified by a number:

- 1..n are the available services
- t=k-1 is the target service
- player k is the scheduler

The game starts with all services in their local initial state.

In each round all services choose a move corresponding to an available operation for its local state. The move of the scheduled player determines the next state.

Composition Problem in ATL

$$\Pi = \{s_{ij} , sch_i, op_a, final_i, last\_sch_i, last\_opt_a\}$$

For each state we build meta-propositions over the set Π:

$state_i$: the local state of the service i

SCH : the index of the scheduled service

LAST_SCH : the index of the previously scheduled service

op : the operation performed by scheduled service

last_opt : the last operation performed by the target

Composition Problem in ATL

For each player $i \in \{1..k\}$ and for each $q \in Q$, the natural number $d_i(q) \geq 1$ is the number of game moves available at state q to player i. In particular we have:

$d_i(q) = |A| \quad \forall i \neq k$

$d_k(q) = k-1$

Note that this doesn't mean that each service i can actually perform exactly |A| operations in each state of TS_i .

Composition Problem in ATL

For each $q,q' \in Q$ we say that q' is a successor of q if exist a move vector $\langle j_1,\ldots,j_k \rangle \in D(q)$ such that $q' = \delta(q, j_1,\ldots,j_k)$.

Being $d_i(q)=|A|$, we can now define a total order over alphabet A obtaining an biunivocal correspondence between services' moves and alphabet operations.

Composition Problem in ATL

For each $q,q' \in Q$ we say that q' is a successor of q if exists a move vector $<j_1,\ldots,j_k> \in D(q)$ such that $q' = \delta(q, j_1,\ldots,j_k)$.

-We now call propositions of q as unprimed, and propositions of q' as primed.

- let be $h=j_k$ (i.e. the index of the scheduled player)
- let $a \in A$ be the operation associated to j_h

Composition Problem in ATL

If the current state is $q \in Q$ and players choose moves $j_1,...,j_k$ then the successor state $q'=\delta(q, j_1,...,j_k)$ is so that:

$sch' = h$

$last\_sch'=sch$

$op'=a$

$state_h'=s \in S_h$ if $<state_h,a,s> \in \delta_h$, otherwise $state_h'=s\_err$

$state'_i=state_i$ $\forall i \neq h, i \in \{1,..,k-1\}$

$last\_opt'=a$ iff $h=t$, otherwise $last\_opt'=last\_opt$

$final_i = true$ iff $state_h' \in F_i$, false otherwise $\forall i \in \{1,..,k-1\}$

Composition Problem in ATL

Checking the existence of an orchestrator is reduced to checking the ATL formula:

$$\ll S_1 \ldots S_n \, , S_k \gg \Box \; ($$

$$\text{Init} \vee \text{state}_t = \text{s\_err} \vee$$

$$(\neg(\text{SCH}=t) \rightarrow \; (\text{opsch}=\text{last\_opt} \wedge$$

$$(\text{final}_t \rightarrow \text{final}_{i \in \{1 \ldots n\}}) \wedge$$

$$\neg(\text{state}_{i \in \{1 \ldots n\}} = \text{s\_err}) \wedge$$

$$\text{LAST\_SCH}=t)$$

$$)$$

$$\wedge$$

$$(\; \text{SCH}=t \rightarrow \neg(\text{LAST\_SCH}=t) \;)$$

$$)$$

Where Init denotes the initial state of the game structure: Init=true iff SCH=null and LAST_SCH=null

Tools

We used two different tools for system specification and verification:

- **MOCHA**: Exploiting modularity in Model Checking
- **MCMAS**: Model Checker for Multi-Agent Systems

jMocha & cMocha

Mocha is available in two versions: jMocha v2.0.1 & cMocha v1.0.1. Both versions offer the following capabilities:

- System specification in **Reactive Module Language**
- System execution (random, guided or mixed mode)
- Implementation verification (refinement)
- Requirement specification and verification:
  - invariant checking
  - **ATL checking** (cMocha only)

Mocha: RML

ReactiveModules is the modeling formalism and input language to Mocha.

The system is described as **atoms** and **modules**.

RML: Atoms

The state of the system is described by a set of state variable: each system state correspond to an assignment of values to the variables.

3 types of variables:

- Controlled by the atom
- Read by the atom
- Awaited (the atom can read their next value)

The behavior of the system consist in:

- An **initial round** which initializes the variables
- An infinite sequence of **guarded update rounds**

RML: Modules

A module is a collection of atoms, containing a declaration of variables that occur in the module:

- Private
- Interface
- External

```
module RandomWalk is
     interface x: (0..9)

     atom IncrDecr
        controls x
        reads x
        init
           [] true -> x' := 0
        update
           [] true -> x' := inc x by 1
           [] true -> x' := dec x by 1
     endatom
endmodule
```

Composition Problem in Mocha

Each player of the game is represented with a module: the scheduler, available services and the target service.

Services are scheduled by the Scheduler at each step: each module representing a service awaits an external variable *sch* controlled by the Scheduler.

Example

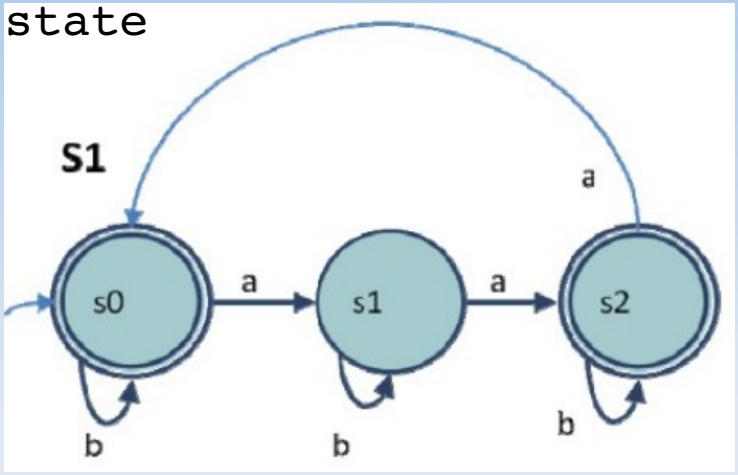
Scheduler

```
type state: {s0,s1,s2,s_err}
type sched_type: {schnil,ST,S1}
type op: {A,B}

module Scheduler
    interface sch : sched_type; last_sch : sched_type
   atom ts
      controls sch,last_sch
      reads sch
      init
         []true ->  sch' := schnil; last_sch':= schnil
      update
         []sch=schnil -> last_sch':=sch; sch':= ST
         []sch=schnil -> last_sch':=sch; sch':= S1
         []sch=ST -> last_sch':=sch; sch':= S1
         []sch=ST -> last_sch':=sch; sch':= ST
         []sch=S1 -> last_sch':=sch; sch':= S1
         []sch=S1 -> last_sch':=sch; sch':= ST
   endatom
endmodule
```

Example

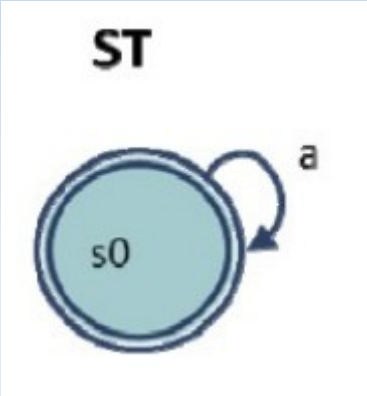
```
module Service1
    interface opS1: op; finalS1:bool; stateS1:state
    external sch: sched_type
    atom ts
        controls stateS1,opS1,finalS1
        reads stateS1,finalS1
        awaits sch
    init
        []true -> stateS1':=s0; finalS1':=true
    update
        []stateS1=s0 & sch'=S1 -> stateS1':=s1; opS1':=A; finalS1':=false
        []stateS1=s0 & sch'=S1 -> stateS1':=s0; opS1':=B; finalS1':=true
        []stateS1=s1 & sch'=S1 -> stateS1':=s2; opS1':=A; finalS1':=true
        []stateS1=s1 & sch'=S1 -> stateS1':=s1; opS1':=B; finalS1':=false
        []stateS1=s2 & sch'=S1 -> stateS1':=s0; opS1':=A; finalS1':=true
        []stateS1=s2 & sch'=S1 -> stateS1':=s2; opS1':=B; finalS1':=true
        []stateS1=s_err & sch'=S1 ->
              stateS1':=s_err; opS1':=A; finalS1':=false
        []stateS1=s_err & sch'=S1 ->
              stateS1':=s_err; opS1':=B; finalS1':=false
        []~(sch'=S1) -> stateS1':= stateS1; finalS1':=finalS1
  endatom
endmodule
```



Example

```
module Target
    interface opST : op; finalST:bool; stateST:state
    external sch: sched_type
    atom ts
        controls stateST,opST,finalST
        reads stateST,finalST,opST
        awaits sch
    init
        []true  ->  stateST' := s0; finalST':=true
    update
        []stateST=s0 & sch'=ST -> stateST':=s0; opST':=A; finalST':=true
        []stateST=s_err & sch'=ST ->
                stateST':=s0; opST':=A; finalST':=false
        []sch'=ST -> stateST':=s_err; opST':=B; finalST':=false
        []~(sch'=ST) -> stateST':=stateST; finalST':=finalST; opST':=opST

    endatom
endmodule
```



Example

A specification file contains a list of specifications. We gave cMocha the ATL specification:

```
atl "formula1" << Scheduler,Service1>> G(
    stateST=s_err |
    (( ~(sch=schnil & last_sch=schnil) & ~(sch=ST)) =>
        ((sch=S1 => opS1=opST) &
        (finalST => finalS1) &
        ~(stateS1=s_err) &
        last_sch=ST )
    )
    &
( sch=ST => ~(last_sch=ST) ));
```

Mocha output: `ATL_CHECK: formula "formula1" failed`

Which is a predictable result, due to the fact that the state Service1.s1 is not a final state.

Mocha : conclusions

Currently the ATL model-checker does not have any mechanism to generate counter-examples.

The impossibility of having any witness or counterexample for ATL checking drove us to search for further confirmations.

MCMAS

MCMAS is a Model Checker for Multi-Agent Systems (MAS).

MCMAS takes in input a MAS specification and a set of formulae to be verified, and it evaluates the truth value of these formulae using algorithms based on Ordered Binary Decision Diagrams (OBDDs).

MCMAS : features

CMAS allows the verification of a number of modalities, including CTL operators, epistemic operators, operators to reason about correct behavior and strategies, with or without fairness conditions.

MCMAS can also be used to run interactive, step-by-step simulations.

Additionally, a graphical interface is provided as an Eclipse plug-in which includes a graphical editor and simulator and graphical analyzer for counterexamples.

MCMAS : ISPL

Multi-Agent Systems are described in MCMAS using a dedicated programming language called **ISPL** (Interpreted Systems Programming Language).

ISPL resembles the SMV language characterizing agents by means of variables and represents their evolution using boolean expressions.

MCMAS : agents

MCMAS distinguishes between two kinds of agents:

- **standard** agents
- **environment** agent

The environment is used to describe boundary conditions and infrastructures shared by standard agents.

MCMAS : agents

In MCMAS each agent (including the environment) is characterized by:

1. A set of **local states**
2. A set of **actions**
3. A rule describing which action can be performed by the agent in a given local state (**protocol**)
4. An **evolution** function, describing how the local states of the agents evolve based on their current local state and on other agents' actions.

Composition problem in MCMAS

- Each player representing a service (either target service or an available service) has been encoded with a standard Agent.

- Environment Agent corresponds to the Scheduler : at each step it sets an observable variable to schedule a player.

- Game moves correspond to Actions: protocol and evolution functions are used to encode the transition system of a service.

Composition problem in MCMAS

Each player chooses an Action at each round, but only the scheduled player can change its current state (hence the global game state) accordingly.

Example

Environment

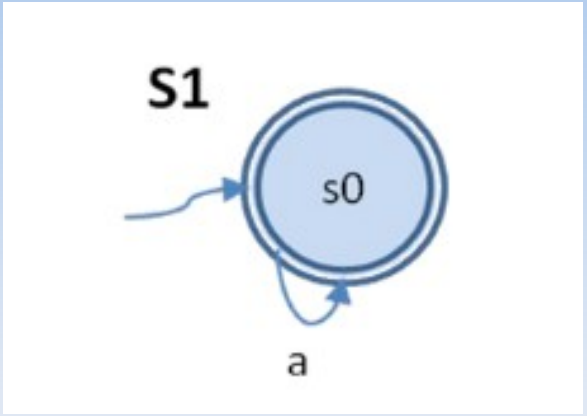
```
Agent Environment
    Obsvars:
        sch : {nil,S1,S2,ST};
    end Obsvars
    Vars:
           lastsch : {nil,S1,S2,ST};
    end Vars
    RedStates:
    end RedStates

    Actions = {nil,S1,S2,ST};

    Protocol:
        Other : {S1,S2,ST}; --It can choose any player in each state.
    end Protocol

    Evolution:
        --sch and lastsch are updated according to the chosen action
        (lastsch = sch) and (sch = S1) if (Action = S1);
        (lastsch = sch) and (sch = S2) if (Action = S2);
        (lastsch = sch) and (sch = ST) if (Action = ST);
     end Evolution
end Agent
```

Example




```
Agent S1

Vars:
stateS1 : {s0,s_err};
finalS1 : boolean;
opeS1 : {a,b};
end Vars

RedStates:
end RedStates

Actions = {a,b}; --service's moves

Protocol:
Other : {a,b}; --moves available in each state
end Protocol
```

Example

Evolution:

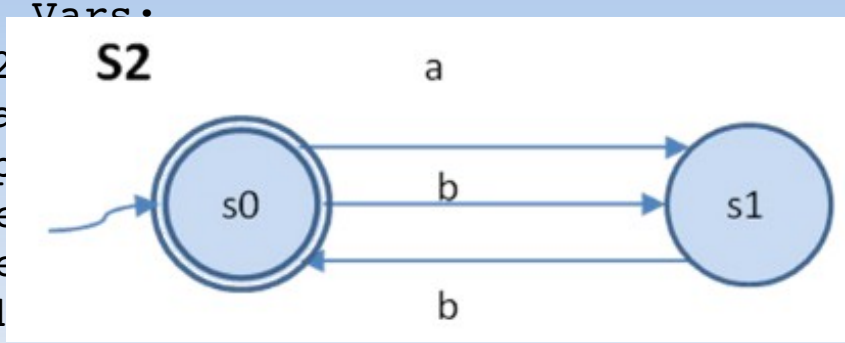
--if S1 has been scheduled and it chooses action a from state s0, then
--the next state will remain s0 with finalS1=true. OpeS1 is set to a.
(stateS1=s0) and (finalS1 = true) and (opeS1=a)
if (stateS1=s0) and (Action=a) and (Environment.Action=S1);

--if the player i chooses a move corresponding to an operation the
--available service it represents cannot actually perform:
(stateS1=s_err) and (opeS1=a) and (finalS1=false)
if (Action=a and stateS1=s_err) and (Environment.Action=S1);
(stateS1=s_err) and (opeS1=b) and (finalS1=false)
if (Action=b) and (Environment.Action=S1);

--if the scheduler chose another service then S1 doesn't evolve even
--if, as requested by the game structure, it can choose an action.
(stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);

end Evolution
end Agent

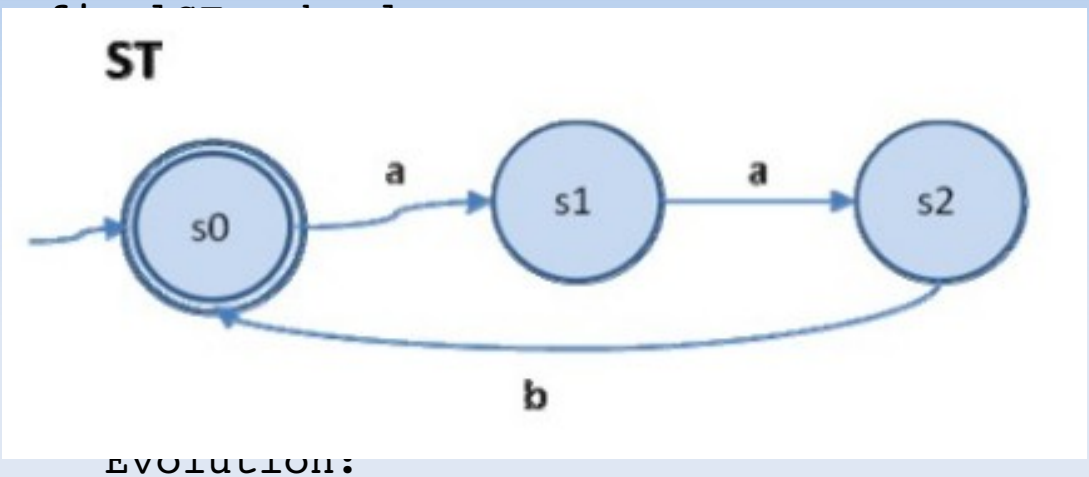
Example




```
Agent S2
   Vars:
stateS2
   fina
     op
      e
     Re
   end
Actions = {a,b};
   Protocol:
  Other : {a,b};
  end Protocol
    Evolution:
      (stateS2=s1) and (finalS2 = false) and (opeS2=b)
   if (stateS2=s0) and (Action=b) and (Environment.Action=S2);
     (stateS2=s1) and (finalS2 = false)  and (opeS2=a)
if (stateS2=s0) and  (Action=a) and (Environment.Action=S2);
      (stateS2=s0) and (finalS2 = true)  and (opeS2=b)
   if (stateS2=s1) and  (Action=b) and (Environment.Action=S2);
      (stateS2=s_err) and (opeS2=a) and (finalS2=false)
   if (Action=a and !(stateS2=s0)) and (Environment.Action=S2);
      (stateS2=s_err) and (opeS2=b) and (finalS2=false)
  if (Action=b and stateS2=s_err) and (Environment.Action=S2);
(stateS2=stateS2) and (finalS2=finalS2) if !(Environment.Action=S2);
                  end Evolution
                  end Agent
```

Example

```
Agent ST
   Vars:
stateST : {s0,s1,s2,s_err};
```




```
Evolution:
    (stateST=s1) and (finalST = false) and (opeST=a)
 if (stateST=s0) and (Action=a) and (Environment.Action=ST);
    (stateST=s2) and (finalST = false) and (opeST=a)
 if (stateST=s1) and (Action=a) and (Environment.Action=ST);
     (stateST=s0) and (finalST = true) and (opeST=b)
 if (stateST=s2) and  (Action=b) and (Environment.Action=ST);
    (stateST=s_err) and (opeST=a) and (finalST=false)
  if (Action=a and stateST=s2) and (Environment.Action=ST);
    (stateST=s_err) and (opeST=a) and (finalST=false)
 if (Action=b and !(stateST=s2)) and (Environment.Action=ST);
(stateST=stateST) and (finalST=finalST) and (opeST=opeST)
                 if !(Environment.Action=ST);
                    end Evolution
                    end Agent
```

Example

```
Evaluation
    TargetMoved if (Environment.sch=ST);
    Replayed if
          ((!(Environment.sch=S1) or (S1.opeS1=ST.opeST)) and
           (!(Environment.sch=S2) or (S2.opeS2=ST.opeST))
          )
          and
          (ST.finalST=false or (S1.finalS1=true and S2.finalS2=true));
    LastTargetMoved if Environment.lastsch=ST;
    Init if (Environment.lastsch=nil and Environment.sch=nil);
   Error if (S1.stateS1=s_err) or (S2.stateS2=s_err);
    Invalid if ST.stateST=s_err;

end Evaluation
```

- **TargetMoved** is evaluated to true in every round in which Target has been scheduled.
- **Replayed** is true if the scheduled agent emulates the last operation performed by the target and if target state is final, then all services are in final state.
- **LastTargetMoved** is evaluated to true if the Scheduler (Environment) chose action ST in the previous round, i.e. if target service has been scheduled in the previous round.
- **Error** and **Invalid** are true available/target services are in error state

Example

```
InitStates
       (S1.stateS1=s0) and (S2.stateS2=s0) and (ST.stateST=s0) and
       (Environment.sch=nil) and (S1.finalS1=true) and (S2.finalS2=true)
       and (ST.finalST=true) and (Environment.lastsch=nil);
end InitStates

Groups
    PlayersPlusEnv = { S1,S2,Environment } ; --excluding agent ST
end Groups
Fairness
end Fairness
```

At the end there is the definition of the ATL formula to be checked. This formula uses the evaluation function previously defined.

```
Formulae
  <PlayersPlusEnv> G (
    Init or Invalid or
    (
        (!TargetMoved -> (Replayed and !Error and LastTargetMoved))
        and
        (TargetMoved -> !LastTargetMoved)
    ));
end Formulae
```

Conclusions

Neither cMocha nor MCMASv0.9.6.2 provide witnesses and we are not currently able to synthesize an orchestrator for the composition problem expressed in ATL.

For this reason our work has been concluded providing extensive examples.