Università degli Studi di Roma “Sapienza”

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

# Composition of stateful deterministic services in Alternating-time Temporal Logic

Seminari di Ingegneria del Software
Metodi Formali nell'Ingegneria del Software

**Paolo Felli**
paolo.felli@gmail.com
**Matteo Vita**
matteo.vita@gmail.com

# 1. Introduction

Web services are self-describing computational elements that support rapid, low-cost and easy composition of loosely coupled distributed applications. From a technical point of view, these services are modular applications that can be described, published, located, invoked and composed over a variety of networks : any piece of code and any application component deployed on a system can be wrapped and transformed into a network available service, by using standard languages and protocols.

The availability of an high level description opens the possibility of composing services in an automatic way, with the aim of realizing target computations.

The promise of Web Service Composition is to use Web Services as fundamental elements for realizing distributed applications and solutions. In particular, when no available services satisfies a desired specification, different available services can be composed and orchestrated in order to realize the specification.

Services can be characterized in different ways, but in this document we follow the approach usually referred to as the Roman Model [1]. In such approach, services are described by their conversational behaviour, modeled as transition systems that capture the possible conversations a service can have with its clients.

In this document, we address the automatic composition of deterministic available services, hence we are referring to services that are fully controllable.

Composition's goal is, given a target service specifying a desired interaction with the client, synthesize an orchestrator generator capable of realizing the target service exploiting the community of available services.

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.

Hence the orchestrator realizes a target service if and only if it's able at every step to delegate every operation executable by the target to one of the available services.

This composition involves two different phases: the composition synthesis, where the specification of an orchestrator is synthesized either manually or automatically, and the composition deployment, that is the actual implementation of the orchestrator specification in a given technology.

Our goal is to verify and realize this composition using Alternating-time Temporal Logic (ATL).

Temporal logic was considered in two varieties : Linear-time Temporal Logic assumes implicit universal quantification over all path that are generated by the execution of a system, while Branching-time Temporal Logic allows explicit existential and universal quantification over all path.

ATL represents a third, more general variety of temporal logic: Alternating-time Temporary Logic offers selective quantification over paths seen as possible outcomes of a game between a system and the environment.

Alternating-time temporal logic have been introduced to enrich temporal logic so that alternating properties can be specified within the logic: since modeling languages for open systems distinguish between internal nondeterminism (choices made by the system) and external nondeterminism (choice made by the environment), a more complex question arises: "can the system resolve its internal choices so that the satisfaction of a property is guaranteed no matter how the environment resolves the external choices?". Alternation can be considered as a natural generalization of existential and universal branching.

We just said that this 'alternating' satisfaction can be viewed as a winning condition in a game between a system and its environment, but in order to capture composition of open systems (i.e systems that interact with their environment and whose behaviors depend on the state of the system as well as the behavior of the environments ) 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.

ATL is interpreted over a concurrent game structure. Every state transition of a concurrent game structure results from a choice of moves, one of each player. The player represent the individual component s and the environment of an open system. Concurrent game structure can capture various forms of synchronous composition for open systems and if augmented with fairness constraints, also asynchronous composition.

We aim to show how a service composition problem instance, for services exporting their behavior in the form of a finite deterministic transition system, 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. Such a requirement can be expressed and verified using alternating-time temporal logic.

This document is organized in three sections as follows: in section 2 we recall the basic framework of our approach including ATL syntax and semantics, in section 3 we formulate the composition problem in ATL logic, finally in the last section we provide implementation examples using tools for system specification and verification: Mocha and MCMAS (v0.9.6.2).

# 2. Premises

## 2.1. Automatic service composition

Services represent software modules capable of performing actions. These modules are intended to interact with a client and their interactions follow a given behavior: according to current state, a service offers a choice of available operations and performs the one chosen by the client, then moves to the corresponding successor state.

Such interaction is potentially infinite and can be stopped by the client whenever the service is left in a consistent state, that is, the service is in a 'final' state.

This interaction pattern requires a formal representation of services; in this approach their (public) dynamic behaviour is described as a finite transition system (TS).

A Transition System TS = $<A, S, s^0, \delta, F>$ is defined as follows:

- A is the finite alphabet of actions
- S is the finite set of states
- $s^0$ is the initial state
- $\delta \subseteq S \times A \times S$ is the transition relation
- F is the set of final states; services can be left only in final states.

We call 'available services' those services that correspond to existing programs and are directly available to the client.

Our goal is, given a target service specifying a desired interaction with the client, to synthesize an orchestrator capable of realizing the target service exploiting the community of available services. Hence the orchestrator realizes a target service if and only if it's able at every step to delegate every operation executable by the target to one of the available services.

Notably the target service itself is represented by a transition system $TS_t = <A, S_t, s0_t, \delta_t, F_t>$ sharing the operations in A, but this service is not one of the available services of the community. Hence, it must be realized by exploiting fragments of the available service behaviors (computations), since these are the only services that correspond to existing programs in the system.

We make these 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 fully controllable, being stateful and deterministic.

## 2.2. **The ATL logic**

We briefly discuss Alternating-time Temporal Logic [2]. ATL is a generalization of the temporal logic CTL and is designed to write requirements of open systems, and is defined by generalizing the existential and universal path quantifiers of CTL.

LTL and CTL are interpreted over Kripke structures, since these structures offer a natural model for computations of closed systems (whose behavior is determined only by the state of the system). 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. For this reason, besides existential and universal requirements, we want to verify if a property can be enforced by the system no matter how the environment resolves its choices.

We already said that this 'alternating' satisfaction can be viewed as a winning condition in a multi-player game between a set Σ of players that represent all the different components involved. This game is equivalent to a 2-players 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 (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. 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 $\Sigma \setminus A$ choose their moves. We say that the ATL formula $\ll A \gg L$ is satisfied in the state q.

$\ll A \gg$ 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. Hence, the existential path quantifier ∃ is equivalent to $\ll \Sigma \gg$, while the universal ∀ corresponds to $\ll \gg$.

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 in which at each step each player chooses a move, and the combination of choices determines a transition from the current state to a successor state.

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, so we are going to avoid the dichotomy between Community transition system and Target transition system: what we aim to obtain is a (single) game structure for the whole set of services.

### Concurrent Game Structures

As defined in [2 - p.6], a Concurrent Game Structure is a tuple $S = < k, Q, \Pi, \pi, d, \delta >$ with the following components:

- A natural number $k \geq 1$ of players. Players are identified with number 1..k.
- A finite set Q of states.
- A finite set Π of propositions (observables).

- For each state $q \in Q$, a set $\pi(q) \subseteq \Pi$ of propositions true at q. The function π is called labeling function.
- For each player $a \in \{1..k\}$ and for each state $q \in Q$, a natural number $d_a(q) \geq 1$ of moves available at state q to player a. We identify the moves available to player a at state q with the number $1,..., d_a(q)$ . For each state $q \in Q$, we write D(q) for the set $\{1..d_1(q)\} \times ... \times \{1..d_k(q)\}$ of move vectors. The function D is called Move Function
- For each state $q \in Q$ and each move vector $\langle j_1,...,j_k \rangle \in D(q)$, a state $\delta(q, j_1,...,j_k) \in Q$ that result from state q if every player $a \in \{1..k\}$ choose move $j_a$

There are three types of game structures for the synchronous composition of open systems:

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

**Moore synchronous** : in every state 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.

The latter case has been used to encode Composition Problem instances as concurrent asynchronous games.

**Turn Based Asynchronous Game Structure**

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. We say that a player $a \in \{1, …,k\}$ is 'scheduled' whenever player k chooses move a. Scheduled player completely determines the next state: moves chosen by other players are 'ignored'.

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 conditions are satisfied:

- $d_k(q) = k-1$
- for all move vectors $\langle j_1,...,j_k \rangle, \langle j_1',...,j_k' \rangle \in D(q)$, if $j_k = j_k'$ and $j_a = j_a'$ for $a=j_k$ then $\delta(q, j_1,...,j_k) = \delta(q, j_1'...,j_k')$

**ATL Syntax and semantics**

The game structure S over whom ATL formulas are interpreted has the same propositions and players as the formula itself: the labeling of the states in S is used to evaluate the atomic formulas of ATL.

For easier reading we resume here ATL syntax and semantics [2 – pp. 15-19].

The temporal logic ATL is defined with respect to a finite set Π of propositions and a finite 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.

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

- q |= p, for propositions p ∈ Π, iff p ∈ π(q).
- q |= ¬ψ iff q |≠ψ.
- 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 haveλ[1] |= ϕ.
- q |= ≪A≫□ϕ iff there exists a set FA 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 $F_A$ 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

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.

# 3. Composition problem in ATL

Given a finite set of services (a community of n available services and a target service) whose behaviours are described by a set of Transition Systems $TS_i = <A_i, S_i, s^0_i, \delta_i, F_i>$, $i \in \{1...n+1\}$, we are going to define a concurrent game structure GS starting from the general definition of Turn-Based Asynchronous Game Structure $S = <k,Q,\Pi,\pi,d,\delta>$.

Premises:

1. We define the alphabet $A = A_1 \cup ... \cup A_{n+1}$ . This is the the global alphabet of operations that all services share.
2. For each service i (available services and target service) we add an error state 's_err' to the set $S_i$. s_err is considered a not-final state.

The Game Structure **GS** is a tuple $< k,Q,\Pi,\pi,d,\delta >$ where:

- $k \geq 3$ is equal to the cardinality of the set of players $\Sigma$. Each player is identified with an integer:
  - $i \in \{1...n\}$ are the available services ($n = k-2$)
  - $t = k-1$ is the target service
  - k is the scheduler

- Q is a finite set of states. The number of states is equal to the powerset of the set $\Pi$ (see below).

- A finite set $\Pi$ of propositions (observables). For each state $q \in Q$, a set $\pi(q) \subseteq \Pi$ of boolean propositions true at q. The function $\pi$ is called labeling function.

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

  - $\mathbf{s_{ij}}$ , $i \in \{1...k-1\}$ $j \in S_i$
  - $\mathbf{sch_i}$ , $i \in \{1...k-1\}$
  - $\mathbf{op_a}$ , $a \in A$
  - $\mathbf{final_i}$ , $i \in \{1...k-1\}$
  - $\mathbf{last\_sch_i}$ , $i \in \{1...k-1\}$
  - $\mathbf{last\_opt_a}$ , $a \in A$

  With this semantics:

$s_{ij}$=true iff service i is in its local state j

$sch_i$=true iff service i was scheduled in the current state (its move determined current state).

$op_a$=true iff the scheduled service performed action a

$final_i$ =true iff service i is in one of its local final states $F_i$

$last\_sch_i$ =true iff service i was the one scheduled in the previous state

$last\_opt_a$=true iff the last action performed (not just chosen) by the target is a.

While these observable propositions have boolean type, we define some more handy meta-propositions for each $q \in Q$:

**$state_i$** : service i is in its (local) state j iff $s_{ij}$ =true and $s_{iy}$ =false $\forall y \neq j$ where $i \in \{1...k-1\}$ $j,y \in S_i$ . We will write $state_i = j$.

**SCH** : We will say that service i was scheduled in the current state (i.e. current state was determined by its move), writing SCH=i , iff $sch_i$ =true and $sch_j$ =false $\forall j \neq i$, $i,j \in \{1...k-1\}$ . If $sch_i$=false $\forall i \in \{1...k-1\}$ then we write SCH=null.

**LAST_SCH** : We will say that LAST_SCH=i iff $last\_sch_i$ =true and $last\_sch_j$ =false $\forall j \neq i$, $i,j \in \{1...k-1\}$. If $last\_sch_i$=false $\forall i \in \{1...k-1\}$ then LAST_SCH=null. Hence LAST_SCH is the player scheduled in the previous round.

**op** : op=a (the operation performed by scheduled player is a) iff $op_a$=true and $op_b$=false $\forall b \neq a$ and $a,b \in A$

**last_opt** : last_opt=a (the last operation performed by the target is a) iff $last\_opt_a$=true and $last\_opt_b$=false $\forall b \neq a$ , $a,b \in A$.

At the beginning of the game we have SCH=null, LAST_SCH=null, $state_i=s^0_i$ $\forall i \in \{1...k-1\}$, i.e. every service is in its local initial state. More precisely we have that, according to the given semantics, (SCH=null $\wedge$ LAST_SCH=null) iff the game is in initial state.

- 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 every state $q \in Q$ a player i selects a move index $1 \leq j_i \leq d_i(q)$.

  In particular, $\forall q \in Q$ we have:

  - $d_i(q)=|A|$ $\forall i \neq k$
  - $d_k(q)=k-1$

This means that the scheduler, as required by the turn-based asynchronous game structure, can always schedule any player of the game, while each service can choose a game move corresponding to any service operation in the alphabet A (note that this doesn't mean that each service i can actually *perform* exactly |A| operations in each state of $TS_i$ ).

As usual, we identify the moves of a player i at state q with the index $1,...,d_i(q)$. For each state $q \in Q$, we write D(q) for the set $\{1..d_1(q)\}$ x ... x $\{1..d_k(q)\}$ of move vectors. The function D is called 'move function'.

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: we'll say that each move index $j_i \leq d_i(q)$ corresponds to a move $a \in A$ according to the total order defined (operation $a \in A$ performed by a player $i \in \{1,..,k-1\}$ representing a service will be the $j_i$-th element of A).

- For each state $q \in Q$ and each move vector $<j_1,...,j_k> \in D(q)$, a state $\delta(q, j_1,...,j_k) \in Q$ that result from state q if every player $i \in \{1..k\}$ choose move $j_i$. Each move vector $<j_1,...,j_k> \in D(q)$ leads to a single successor state $\delta(q, j_1,...,j_k)$.

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

  - we 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 $j_h$ , that is the move index chosen by player h (as we said this operation is univocally determined).

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

  - SCH'=h

    i.e. $sch'_h$=true and $sch'_j$=false $\forall j \neq h$, $j \in \{1...k-1\}$

  - LAST_SCH'=SCH

    i.e. $last\_sch'_i = last\_sch_i$ $\forall i \in \{1...k-1\}$

  - op'=a

    i.e. $op'_a$=true and $op'_b$=false $\forall$ $b \neq a$, $a,b \in A$

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

i.e. $s'_{hs}$ =true and $s'_{hy}$ =false $\forall y \neq s$ $(y,s \in S_i)$ if player h can perform operation a in its local state $state_h$ according to $TS_i$ : $state_h \rightarrow_a s$

otherwise $s'_{hs\_err}$ =true and $s'_{hy}$ =false $\forall y \neq s\_err$ $(y,s\_err \in S_i)$

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

  i.e. $s'_{ij} = s_{ij}$ $\forall i \neq h$ , $i \in \{1...k-1\}, j \in S_i$

- last_opt'=a iff h=t, otherwise last_opt'=last_opt

  i.e. $last\_opt'_a$=true and $last\_opt'_b$=false $\forall b \neq a$ with $a,b \in A$ if target has been scheduled, otherwise $last\_opt'_a = last\_opt_a$ $\forall a \in A$.

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

Note how whenever a scheduled service (available or target service) represented by a player $i \in \{1,..,k-1\}$ selects a move index corresponding to an operation it cannot actually perform in its local state, then $state_i$=s_err in the successor state q'. Note that there is no way the proposition $s_{i,s\_err}$ can change its value from true to false since s_err has no outbound edges in $ST_i$ .

Note that Turn-base asynchronous game structure constraint is respected:

For all move vectors $<j_1,...,j_k>,<j_1',...,j_k'> \in D(q)$, if $j_k = j_k'$ and $j_h = j_h'$ for $h=j_k$ then $\delta(q, j_1,...,j_k) = \delta(q, j_1'...,j_k')$

This game structure follows turn-based asynchronous game structure requirements, granting observability and controllability of available services as required by assumptions.

**ATL formula**

We show now how we intend to reduce the search for a composition to the search for winning strategies for the corresponding multi-player game over structure GS.

Referring to ATL semantics, a composition exists iff the players representing the available services and the scheduler can always cooperate, irrespective of how the target service chooses its moves, to enforce computations that satisfy a formula, interpreted over the set of observable propositions Π, capturing the requirements of the Composition Problem.

More precisely this corresponds to the existence a set $F_A$ of strategies, one for each player in A={1..n,k} such that, starting from the initial game state q, for all the resulting computations $\lambda \in out(q, F_A)$ and all positions $i \geq 0$ we have that $\lambda[i]$ satisfy the given formula.

- The basic idea is to accept computations in which the scheduler schedules players alternating between target and available services:

  $( \neg(SCH=t) \rightarrow LAST\_SCH=t ) \wedge (SCH=t \rightarrow \neg(LAST\_SCH=t))$

- Whenever an available service is scheduled, we require that the player can repeat the last operation performed by the scheduler (player t) without ending up in an error state:

  $\neg(SCH=t) \rightarrow (opsch=last\_opt \wedge \neg(state_{i\in\{1\ldots n\}}=s\_err) )$

- Whenever the target service moves to a final state, than all services have to be in a local final state in the next game state:

  $\neg(SCH=t) \rightarrow ( final_t \rightarrow final_{i\in\{1\ldots n\}})$

- When the player t is scheduled, it has free choice of action: it is able to select any possible move, but we are not interested in computations in which the target chooses move indexes corresponding to operations it can't actually perform according to its transition system. We want to check that all this requirements can be enforced by players in $\Sigma\backslash\{t\}$ (i.e. irrespective of target choices), so we simply accept all game states in which the target selects an invalid move; this can be done because, whenever target service is scheduled, all branches will be explored in order to check the formula.

Checking the existence of an orchestrator is therefore reduced to checking the this ATL formula w.r.t. the game structure GS and the set of players (we call here $S_i$ the player i).

$$
\begin{aligned}
&\ll S_1 \ldots S_n, S_k \gg \Box \; ( \\
&\quad Init \;\vee\; state_t=s\_err \;\vee \\
&\quad ( \\
&\qquad ( \neg(SCH=t) \rightarrow ( opsch=last\_opt \wedge \\
&\qquad\qquad ( final_t \rightarrow final_{i\in\{1\ldots n\}} ) \wedge \\
&\qquad\qquad \neg(state_{i\in\{1\ldots n\}}=s\_err) \wedge \\
&\qquad\qquad LAST\_SCH=t )) \\
&\qquad \wedge \\
&\qquad ( (SCH=t) \rightarrow \neg(LAST\_SCH=t) ) \\
&\quad ))
\end{aligned}
$$

Where Init denotes the initial state of the game structure: Init=true iff sch=null and last_sch=null (and Init=true $\rightarrow state_i=s^0_i \;\; \forall\, i\in\{1\ldots k-1\}$ ).

# 4. Implementation

## 4.1. jMocha & cMocha [3]

Mocha is a growing interactive software environment for system specification and verification. The main objective of Mocha is to exploit, rather than destroy, design structure in automatic verification. Mocha is intended as a vehicle for development of new verification algorithms and approaches. MOCHA is available in two versions, cMocha (version 1.0.1) and jMocha (version 2.0).

Both versions offer the following capabilities:

- ✔ System specification in the language of *Reactive Modules* (Reactive Module Language: RML[4]). Reactive Modules allow the formal specification of heterogeneous systems with synchronous, asynchronous, and real-time components. Reactive Modules support modular and hierarchical structuring and reasoning principles.
- ✔ System execution by randomized, user-guided, or mixed-mode trace generation
- ✔ Requirement specification. Mocha's checker can perform invariant-checking: an invariant of a module is a predicate that is intended to hold true in all reachable states of the module. Mocha supports the checking of state invariants as well as transition invariants (boolean formulae which involve both current and next state variables). Invariants are expressed by judgements.

  In addition to invariant checking, cMocha supports also ATL requiremens. The logic ATL allows the formal specification of requirements that refer to collaborative as well as adversarial relationships between modules.
- ✔ Requirement verification by ATL-model checking (cMocha only) and both symbolic and enumerative model checking.
- ✔ Implementation verification by checking trace containment between implementation and specification modules.

We used jMocha just for tests and simulations (random simulation, manual simulation, and game simulation) but its lack of an ATL checker drove our attention on cMocha.

### cMOCHA: Reactive Modules

ReactiveModules is the modeling formalism and input language to Mocha. It provides extensive facilities for the modular description of a system, and for modeling both synchronous and asynchronous types of behavior.

An input file for MOCHA is a file with the .rm extension, which is what mocha assumes by default. The system is described as atom and modules.

**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. The behavior of the system consist in an initial round which initializes the variables to their initial values, followed by an infinite sequence of update rounds, which assign new values to the

variables, thus describing the evolution of the system's state. You can also think of the initial round correspond to the initial states of the transition system, and the update rounds define the transition relation.

You can access the next value of the variable x (*latched value*) just typing x' (*updated value*).

Atoms and modules are used to specify the initial and update rounds for all variables. An atom is the basic unit used to described the initial condition and transition relation of a group of related variables. It has three types of variables:

- **Controlled variables**: the variables for which the atom can specify the values in each round. Each variable is controlled by almost one atom
- **Read variables**: the variables whose current value can be read by the atom to decide the next values of the controlled variable
- **Awaited variables**: The variables whose next value can be read by the atom in order to decide the next value of the controlled variable

The Guarded Command statements following the *init* keyword specify the values of the controlled variables at the end of the initial round. The guarded command statements following the update keyword specify the values of the controlled variables at the end of an update round. A guarded command consist in two part : a guard that is Boolean expression specifying when the guarded command can be executed, and a list of assignments, used to specify the next value of the controlled variables.

This is an example of an atom:

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

The atom specifies that the variable x has initially value 0 and that this value can be non-deterministically incremented or decremented by 1 at each round, being the guards of all guarded commands always true.

The smallest units of input are modules, not atoms, so you cannot feed an atom as input to Mocha but it has to be embedded in a module.

**MODULES** : A module is a collection of atoms, together with a declaration of variables that occur in the module. There are 2 types of module

1. Simple modules, obtained bay specifying directly the atoms composing the module. A simple module is defined with the construct module module-name is module-body
2. Composite module, obtained by combining or modifying existing modules.

Associated with each module are three set of variables:

- **Private** variables are the variables that are controlled by some atom of the module, and that cannot be read or awaited by other modules. The values of a private variables is thus local to the module.
- **Interface** variable are the variables that are controlled by some atom in the module, and that can be read or awaited by atoms in other modules. These variables cannot however be controlled by atoms of other modules, according to the general rule stating that a variable can be controlled by at most one atom.
- **External** variables are the variables whose value can be read or awaited by the atoms in the module. These module cannot be controlled by any atom in the module.

Each of the private and interface variables of the module must be controlled by some atom, as shown in the example:

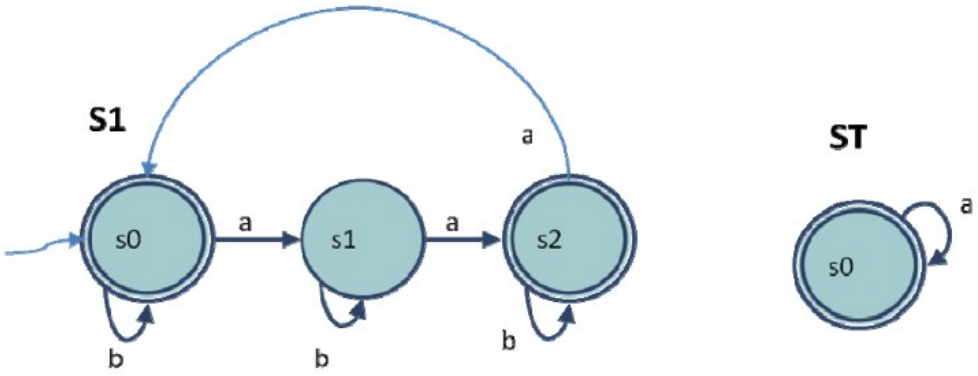
```
module RandomWalk is
      interface x: (0..9)

      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
```

This module is a valid input to Mocha.

### Composition problem in cMOCHA

Now we introduce a representation of the Service Composition's problem using cMOCHA. Each player is represented through a module: scheduler, available services and 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.

We show here a basic example with a target service (ST) looping in the same state with operation 'a' and a single available service (S1).




We defined custom enumeration types for some variables due to the fact that Mocha sometimes incurs a segmentation fault error when using integer enumeration types.

In "*example1.rm*" file we write:

```
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

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

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
```

```
Main:= Service1 || Target || Scheduler
```

Turn-based Asynchronous Game Structure requirements are respected: at each round the Scheduler updates the value of the *sch* variable nondeterministically and the move chosen by scheduled player completely determines the next state. All atoms are deterministic.

Note that every times the Target service is not scheduled, its operation remains unchanged: this is the reason why, in the ATL formula, Target.opST is used in the place of last_opt (the operation chosen by the target at the 'previous round', as defined in GS game structure).

Through parallel composition the three modules have been combined into a single module *Main* whose behavior captures the interaction between them. Such a composition is possible as long as simple modules are compatible. Two simple modules P and Q are compatible if the set of interface variables of P and Q are disjoint and the global 'awaits relation' is acyclic: modules cannot be in a deadlock.

A specification file contains a list of specifications, i.e. invariant or ATL formulae. In our case the specification is given through the ATL formula *formula1* in the "*example1.spec*" file:

```
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) )

);
```

ATL-check is executed typing the following command in cMocha command interface:

```
atl_check Main formula1
```

And this is the output generated:

```
Converting formula to existential normal form...
Performing semantic check on the formulas...
SIM: building atom dependency info
Start model checking...
Building transition relations for module...
Ordering variables using sym_static_order
Transition relation computed : 3 conjuncts
Calling Dynamic Reordering with sift
Done initializing image info...
Building the initial region of the module...
Model-checking formula "formula1"
ATL_CHECK: formula "formula1" failed
```

Which is a predictable result, due to the fact that the state Service1.s1 is not a final state. If we force state s1 to be final, cMocha produces the same output ending with:

```
ATL_CHECK: formula "formula1" passed
```

### cMocha: conclusions

Currently the ATL model-checker does not have any mechanism to generate counter-examples. Despite what's stated in cMocha manual [3 – p.72], the original plan to integrate model checker to provide counter-examples and witnesses is currently suspended by developers since the project is no more supported.

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

## 4.1. MCMAS

This section is mainly extracted by [5]. We used MCMAS v0.9.6.2 and v0.9.7.1.

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).

Whenever possible, MCMAS produces counterexamples for false formulae and witnesses for true formulae (but MCMAS doesn't provide this feature for ATL formulas yet).

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 with syntax recognition, a graphical simulator, and a graphical analyzer for counterexamples.

**System of agents**

Multi-Agent Systems are described in MCMAS using a dedicated programming language derived from the formalism of interpreted systems. This language, called ISPL (Interpreted Systems Programming Language), resembles the SMV language characterizing agents by means of variables and represents their evolution using boolean expressions.

MCMAS distinguishes between two kinds of agents: "standard" agents, and the environment agent. The environment is used to describe boundary conditions and infrastructures shared by standard agents and it is modeled similarly to standard agents. In brief, 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 an agent in a given local state. We call this rule a 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.

**Local states**. Local states are defined in terms of local variables (i.e. corresponding to all the possible combinations of their values). Local states are private and each agent can observe only its own local states, and all the other parameters discussed below (protocol and evolution function) cannot refer to other agents' local variables.

The only exception is the environment agent: for this agent two kind of variables can be defined: standard variables and observable variables. Standard agents can 'peek' at the observable variables of the environment and their evolution function can refer to these variables.

**Actions**. Each agent (including the environment) is allowed to perform actions. It is assumed that all actions performed are visible by all the other agents.

**Protocols**. Protocols describe which actions can be performed in a given local state. As local states are defined in terms of variables, the protocol for an agent is expressed as a function from variable assignments to actions.

ISPL protocols are not required to be exhaustive: it is sufficient to specify only the variables assignments relevant to the execution of certain actions, and introduce a

catch-all assignment by means of the keyword 'Others'. Protocols are not required to be deterministic.

**Evolution functions**. The evolution function for an agent describes how variable assignments change as a results of the actions performed by all the other agents.

Fairness conditions can also be specified in ISPL, to rule out unwanted behaviour.

### ISPL overview: syntax

A multi-agent system specified in ISPL is composed of an Environment agent and a set of standard agents. Each agent has a set of local variables and the Environment also has a set of observable variables, which can be "observed" by other agents.

### Definition of variables

Currently, ISPL allows three types of variables: Boolean, enumeration and bounded integer. Suppose x, y and y are variables of Boolean, enumeration and bounded integer respectively. They are be defined as follows:

x : boolean;  y : {a, b, c};  z : 1 .. 4;

A comparison over Boolean variables or enumeration variables can only be an equality test.

### Definition of actions

All actions of an agent are defined in the section Actions:

Actions = { a1, b2, c3};

### Definition of protocol function

A line in a protocol function is composed of a condition, which is a Boolean formula over local states, and a list of actions. The condition represents all local states that satisfy the condition and the list of actions allowed to be performed in local states specified by the condition. In this example :

x = true and z < 2 : { a1, a3};

x = true and z < 2 is the condition and {a1, a3} is the list of actions.

The conditions appearing in different lines do not need to be mutually exclusive, i.e., the conjunction of these two conditions needs not to be false. If this is the case, the agent has nondeterministic behavour and all behaviours are considered possible by MCMAS. For an agent that has many local states, it might be unrealistic or even impossible to specify actions for every state. MCMAS includes the reserved keyword 'Other'; e.g. Other : { action-list };

### Definition of evolution function

A line in an evolution function consists of a set of assignments of local variables (and observable variables for the Environment) and an enabling condition, which is a

Boolean formula over local variables, observable variables of the Environment, and actions of all agents.

The left hand side (LHS) of an assignment is a local/observable variable being assigned to a new value and the right hand side (RHS) is a truth value or a Boolean local/observable variable if LHS is a Boolean variable, an enumeration value or an enumeration local/observable variable if LHS is an enumerate variable, or an arithmetic expression if LHS is a bounded integer variable.

An observable variable must have a prefix "Environment", such as Environment.x . Multiple assignments can be connected by the keyword 'and'. In an enabling condition, all observable variable must have the prefix "Environment". A proposition over actions is of the form XXX.Action = xxx, where XXX is the name of an agent and xxx is one of its actions. This is a possible line of an evolution function:

(x = true and z = Environment.z + 1) if (y = b and TestAgent.Action = a1);

This is read as: "in the next step, the value of x is true and the value of z is equal to the (current) value of z for the Environment if the current value of y is b and TestAgent is performing action a1".

**Definition of evaluation function**

An evaluation function consists of a group of atomic propositions, which are defined over global states. Each atomic proposition is associated with a Boolean formula over local variables of all agents and observable variables in the Environment. The proposition is evaluated to true in all the global states that satisfy the Boolean formula. Every variable involved in the formula has a prefix indicating the agent the variable belongs to. An example of defining an atomic proposition is shown below:

happy if Environment.x = true and TestAgent.z < Environment.z;

**Definition of initial states**

Initial states are defined by a Boolean formula over variables. For simplicity, arithmetic expressions are not allowed. Below is an example:

Environment.x=false and Environment.y=a;

**Definition of groups**

Groups are used in formulae involving group modalities. A group includes one or more agents, including the Environment, such as

group1 = { TestAgent, Environment };

**Definition of formulas to be checked**

A formula to be verified is defined over atomic proposition. It can have one of the following forms:

formula ::= ( formula ) | formula and formula | formula or formula | ! formula | formula –> formula | AG formula | EG formula | AX formula | EX formula | AF formula | EF formula | A ( formula U formula ) | E ( formula U formula ) | K ( AgentName ,

formula ) | GK ( GroupName , formula ) | GCK ( GroupName , formula ) | O ( AgentName , formula ) | KH ( AgentName , AgentNameOrGroupName , formula ) | DKH ( GroupName , AgentNameOrGroupName , formula ) | DK ( GroupName , formula ) | < GroupName > X formula | < GroupName > F formula | < GroupName > G formula | < GroupName > ( formula U formula ) | AtomicProposition

**Using MCMAS for the composition problem**

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 player. Game moves correspond to Agents' Actions.

There are two environment variables: variable *sch* which is observable by all agents and local variable *lastsch*. The first one is used to select nondeterministically scheduled players at each round, while *lastsch* holds the value of *sch* at previous round.

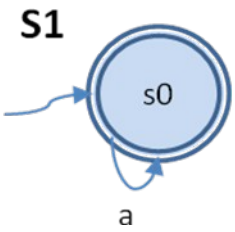
```
Agent Environment
      Obsvars:
            sch : {nil,S1,S2,ST};
      end Obsvars
      Vars:
            lastsch : {nil,S1,S2,ST};
      end Vars
      RedStates:
      end RedStates

      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
```

The protocol section defines the list of actions allowed to be performed in local states. We can notice how, as stated in the Turn-based asynchronous game structure, in every state the scheduler has the possibility to choose either the Target (ST) or one of the community available services (S1, S2).

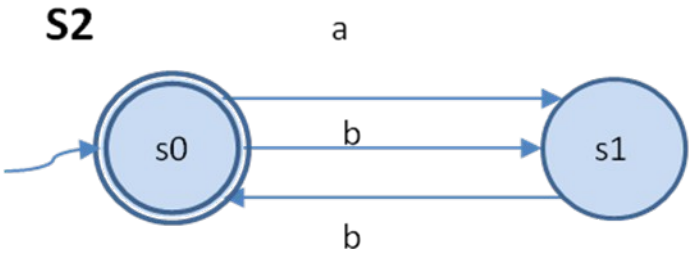
Now we show the Agents representing remaining players, i.e. available services and target service.

Each Agent has a variable for the current state, one boolean variable that indicates if the corresponding service is in a (local) final state and, as for the Scheduler, a variable holding the chosen action.

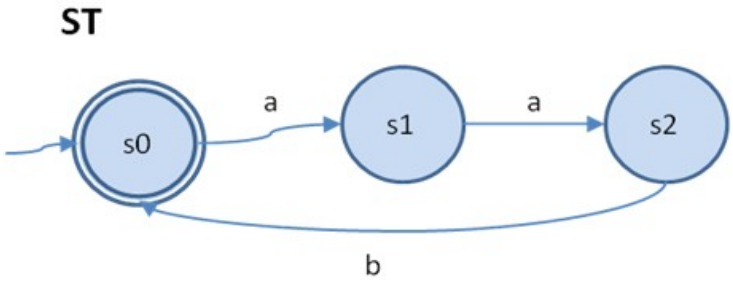
```
Agent S1

    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

    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
```



```
Agent S2
    Vars:
        stateS2 : {s0,s1,s_err};
        finalS2 : boolean;
        opeS2 : {a,b};
    end Vars
    RedStates:
    end RedStates
    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
```




```
Agent ST
    Vars:
        stateST : {s0,s1,s2,s_err};
        finalST : boolean;
        opeST : {a,b};
    end Vars
    RedStates:
    end RedStates

    Actions = {a,b};

    Protocol:
        Other : {a,b};
     end Protocol

     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 or stateST=s_err))
                 and (Environment.Action=ST);
        (stateST=s_err) and (opeST=b) 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
```



As was done in cMocha, ST's action remains unchanged every time ST is not scheduled. Hence opeST variable always holds the last operation chosen by the Target. Here we show the code of the evaluation function:

```
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 (The environment performed Action ST).

**Replayed** is true if the scheduled agent replicated the last operation performed by the target and all services are in final state if target state is final.

**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.

**Init** is true if the game is in its initial state.

**Error** is evaluated to true whenever an available service is in its local error state.

**Invalid** is true if the agent ST is in error state.

```
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
```

**MCMAS results**

As said before, the current version of MCMAS doesn't produce witnesses or counterexample for ATL formulas, so MCMAS is able to give us a simple boolean verification result. For the previous example it was TRUE.

## 4.3 Further examples

Not being able to generate an orchestrator or witnesses, we provide here some examples. All examples are provided with a variant showing how the verification result changes accordingly. Examples' code follows strictly the one shown above.

These examples are ordered by increasing complexity.

**References**

**[1]** *D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, F. Patrizi.* Automatic Service Composition and Syntesis: the Roman Model.

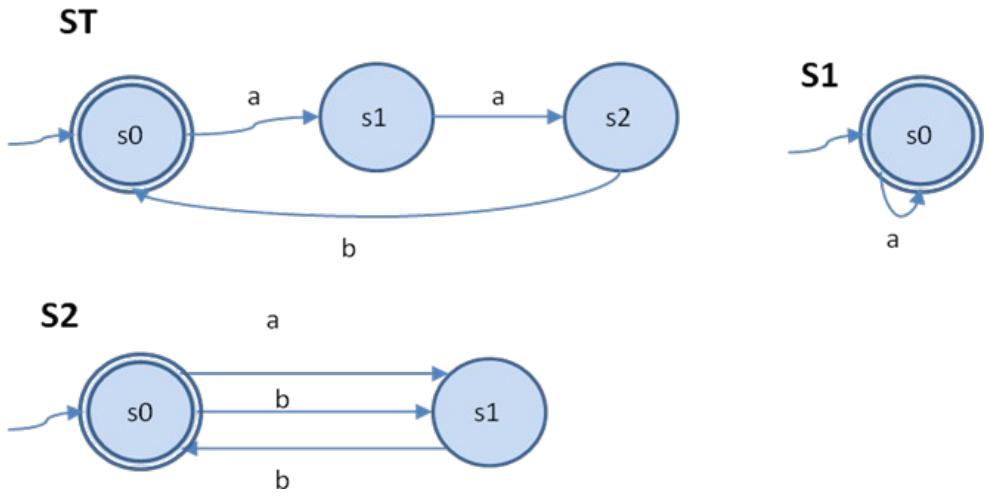
**[2]** *R. Alur, T. A. Henziger, O. Kupferman.* Alternating-time Temporal Logic.

**[3]** *A. Alur, H. Anand, R. Grosu, L. de Alfano, T.A. Henziger, B. Horowits et al.* Mocha 1.0.1 User Manual. cMocha and jMocha documentation and downloads can be found at http://mtc.epfl.ch/software-tools/mocha/

**[4]** *R. Alur, T. A. Henziger.* Computer-Aided Verification, Chapter 1.

**[5]** MCMAS v0.9.6.2 User Manual. MCMAS homepage at: http://www-lai.doc.ic.ac.uk/mcmas/

**Example 1**



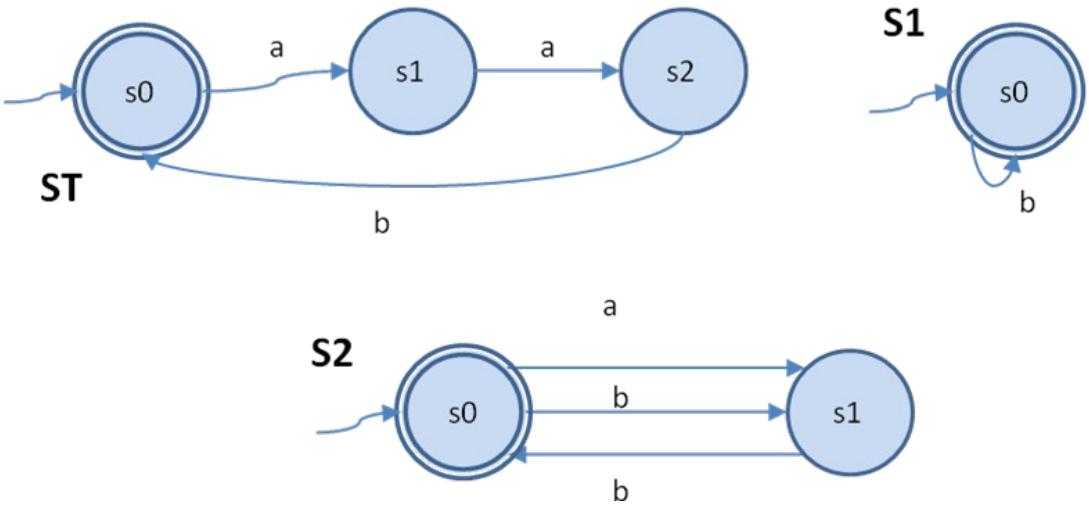

We can see how in every state Service S2 can perform operation b and S1 can loop with operation a. MCMAS result is: TRUE

```
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};
    end Protocol
    Evolution:
            (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

Agent S1
    Vars:
        stateS1 : {s0,s_err};
        finalS1 : boolean;
        opeS1 : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
```

```
    end Protocol
    Evolution:
      (stateS1=s0) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
      (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);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

Agent S2
    Vars:
        stateS2 : {s0,s1,s_err};
        finalS2 : boolean;
        opeS2 : {a,b};
    end Vars
    RedStates:
    end RedStates
    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

Agent ST
    Vars:
        stateST : {s0,s1,s2,s_err};
        finalST : boolean;
        opeST : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
    end Protocol
    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 or stateST=s_err)) and
               (Environment.Action=ST);
      (stateST=s_err) and (opeST=b) 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

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

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 } ;
end Groups

Fairness
end Fairness

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

**Example 1b**



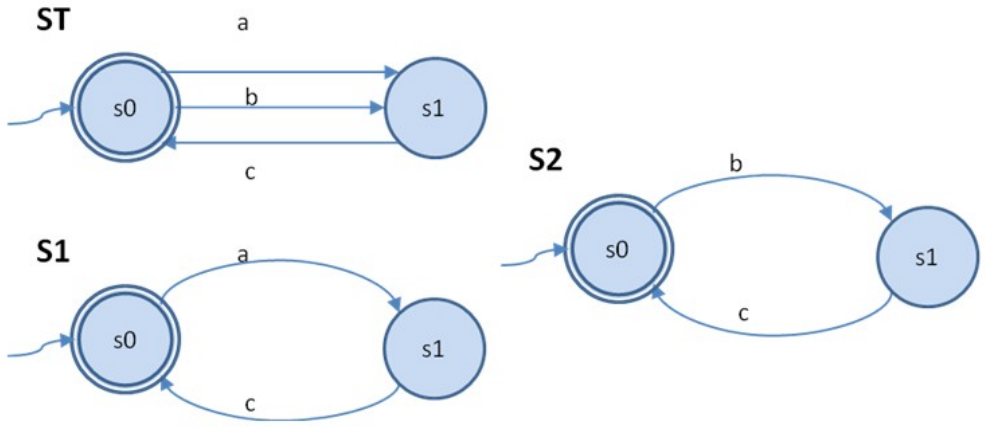

There's no way operation a can be performed twice by S1 or S2. MCMAS result: FALSE

```
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};
    end Protocol
    Evolution:
            (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

Agent S1
    Vars:
        stateS1 : {s0,s_err};
        finalS1 : boolean;
        opeS1 : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
    end Protocol
    Evolution:
        (stateS1=s0) and (finalS1 = true) and (opeS1=b)
```

```
            if (stateS1=s0) and (Action=b) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=a) and (finalS1=false)
            if (Action=a) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=b) and (finalS1=false)
            if (Action=b and stateS1=s_err) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

Agent S2
    Vars:
        stateS2 : {s0,s1,s_err};
        finalS2 : boolean;
        opeS2 : {a,b};
    end Vars
    RedStates:
    end RedStates
    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=a) 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

Agent ST
    Vars:
        stateST : {s0,s1,s2,s_err};
        finalST : boolean;
        opeST : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
    end Protocol
    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 or stateST=s_err)) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=b) 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

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

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 } ;
end Groups

Fairness
end Fairness

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

**Example 2**



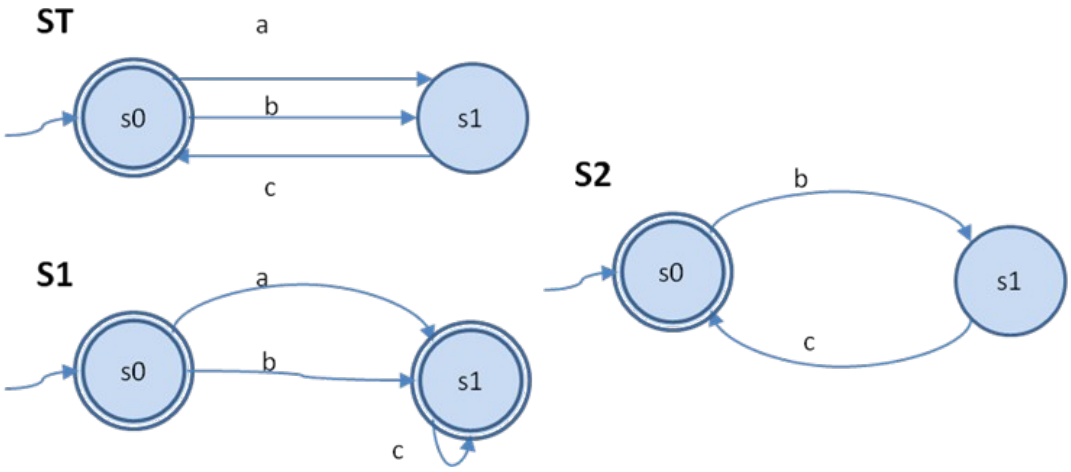

Depending on the operation chosen by ST at s0, service S1 or S2 can be easily used to replicate the target. MCMAS result: TRUE

```
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};
    end Protocol
    Evolution:
            (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

Agent S1
    Vars:
        stateS1 : {s0,s1,s_err};
        finalS1 : boolean;
        opeS1 : {a,b,c};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b,c};
    Protocol:
        Other : {a,b,c};
    end Protocol
    Evolution:
```

```
      (stateS1=s1) and (finalS1 = false) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
      (stateS1=s0) and (finalS1 = true)  and (opeS1=c)
            if (stateS1=s1) and  (Action=c) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=a) and (finalST=false)
            if (Action=a and !(stateS1=s0)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=b) and (finalS1=false)
            if (Action=b) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=c) and (finalS1=false)
            if (Action=c and !(stateS1=s1)) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

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

Agent ST
    Vars:
        stateST : {s0,s1,s_err};
        finalST : boolean;
        opeST : {a,b,c};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b,c};
    Protocol:
```

```
          Other : {a,b,c};
    end Protocol
    Evolution:
      (stateST=s1) and (finalST = false) and (opeST=a)
            if (Action=a and stateST=s0) and (Environment.Action=ST);
      (stateST=s1) and (finalST = false) and (opeST=b)
            if (Action=b and stateST=s0) and (Environment.Action=ST);
      (stateST=s0) and (finalST = true) and (opeST=c)
            if (Action=c and stateST=s1) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=a) and (finalST=false)
            if (Action=a and !(stateST=s0)) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=b) and (finalST=false)
            if (Action=b and !(stateST=s0)) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=c) and (finalST=false)
            if (Action=c and !(stateST=s1)) and (Environment.Action=ST);
      (stateST=stateST) and (finalST=finalST) and (opeST=opeST) if !
(Environment.Action=ST);
      end Evolution
end Agent

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

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 } ;
end Groups
Fairness
end Fairness

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

**Example 2b**



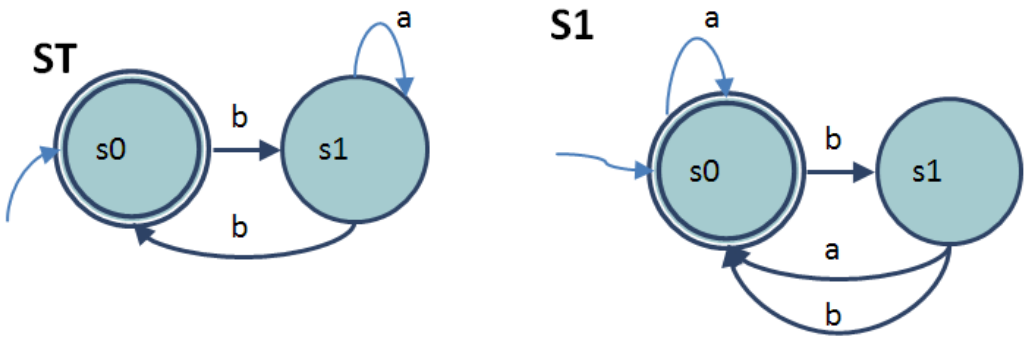

S1 has been changed. MCMAS result: FALSE

```
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};
    end Protocol
    Evolution:
            (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

Agent S1
    Vars:
        stateS1 : {s0,s1,s_err};
        finalS1 : boolean;
        opeS1 : {a,b,c};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b,c};
    Protocol:
            Other : {a,b,c};
    end Protocol
    Evolution:
```

```
      (stateS1=s1) and (finalS1 = false) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
      (stateS1=s1) and (finalS1 = false) and (opeS1=b)
            if (stateS1=s0) and (Action=b) and (Environment.Action=S1);
      (stateS1=s1) and (finalS1 = true)  and (opeS1=c)
            if (stateS1=s1) and  (Action=c) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=a) and (finalS1=false)
            if (Action=a and !(stateS1=s0)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=b) and (finalS1=false)
            if (Action=b and !(stateS1=s0)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=c) and (finalS1=false)
            if (Action=c and !(stateS1=s1)) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

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

Agent ST
    Vars:
        stateST : {s0,s1,s_err};
        finalST : boolean;
        opeST : {a,b,c};
    end Vars
    RedStates:
    end RedStates
```

```
    Actions = {a,b,c};
    Protocol:
          Other : {a,b,c};
    end Protocol
    Evolution:
      (stateST=s1) and (finalST = false) and (opeST=a)
            if (Action=a and stateST=s0) and (Environment.Action=ST);
      (stateST=s1) and (finalST = false) and (opeST=b)
            if (Action=b and stateST=s0) and (Environment.Action=ST);
      (stateST=s0) and (finalST = true) and (opeST=c)
            if (Action=c and stateST=s1) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=a) and (finalST=false)
            if (Action=a and !(stateST=s0)) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=b) and (finalST=false)
            if (Action=b and !(stateST=s0)) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=c) and (finalST=false)
            if (Action=c and !(stateST=s1)) and (Environment.Action=ST);
      (stateST=stateST) and (finalST=finalST) and (opeST=opeST)
            if !(Environment.Action=ST);
      end Evolution
end Agent

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

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 } ;
end Groups
Fairness
end Fairness

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

**Example 3**



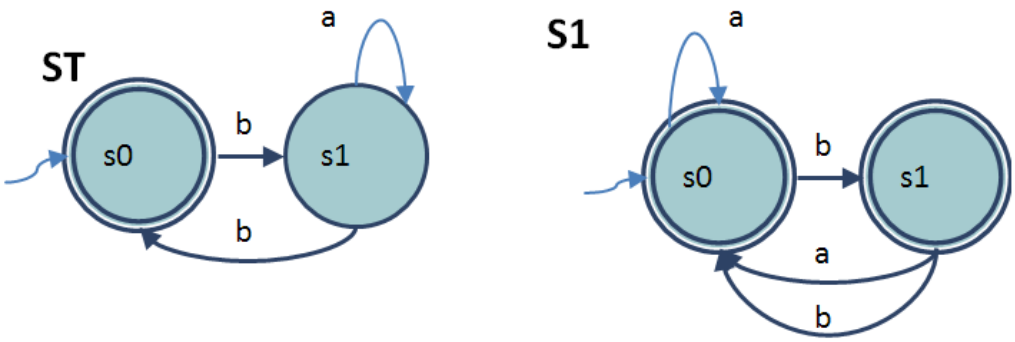

S1 cannot simulate service ST. Since S1.s1 in not final, the sequence of actions b,a,b brings ST in a final state leaving S1 in a not-final state. MCMAS result: FALSE

```
Agent Environment
      Obsvars:
            sch : {nil,S1,ST};
      end Obsvars
      Vars:
            lastsch : {nil,S1,ST};
      end Vars
      RedStates:
      end RedStates
      Actions = {nil,S1,ST};
      Protocol:
            Other : {S1,ST};
      end Protocol
      Evolution:
            (lastsch = sch) and (sch = S1) if (Action = S1);
            (lastsch = sch) and (sch = ST) if (Action = ST);
      end Evolution
end Agent

Agent S1
      Vars:
            stateS1 : {s0,s1,s2,s_err};
            finalS1 : boolean;
            opeS1 : {a,b};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b};
      Protocol:
            Other : {a,b};
      end Protocol
      Evolution:
```

```
        (stateS1=s0) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
        (stateS1=s1) and (finalS1 = false) and (opeS1=b)
            if (stateS1=s0) and (Action=b) and (Environment.Action=S1);
        (stateS1=s0) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s1) and (Action=a) and (Environment.Action=S1);
        (stateS1=s0) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s1) and (Action=b) and (Environment.Action=S1);
      (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 stateS1=s_err) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

Agent ST
      Vars:
            stateST : {s0,s1,s_err};
            finalST : boolean;
            opeST : {a,b};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b};
      Protocol:
            Other : {a,b};
      end Protocol
      Evolution:
        (stateST=s1) and (finalST = false) and (opeST=b)
            if (stateST=s0) and (Action=b) and (Environment.Action=ST);
        (stateST=s1) 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=s1) and (Action=b) and (Environment.Action=ST);
        (stateST=s_err) and (opeST=a) and (finalST=false)
            if (Action=a and !(stateST=s1)) and (Environment.Action=ST);
        (stateST=s_err) and (opeST=a) and (finalST=false)
            if (Action=b and stateST=s_err) and (Environment.Action=ST);
        (stateST=stateST) and (finalST=finalST) and (opeST=opeST)
            if !(Environment.Action=ST);
      end Evolution
end Agent

Evaluation
      TargetMoved if (Environment.sch=ST);
      Replayed if ((!(Environment.sch=S1) or (S1.opeS1=ST.opeST)) and
                  (ST.finalST=false or S1.finalS1=true));
```

```
      LastTargetMoved if Environment.lastsch=ST;
      Init if (Environment.lastsch=nil and Environment.sch=nil);
      Error if (S1.stateS1=s_err);
      Invalid if ST.stateST=s_err;
end Evaluation

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

Groups
      PlayersPlusEnv = { S1,Environment } ;
end Groups

Fairness
end Fairness

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

**Example 3b**



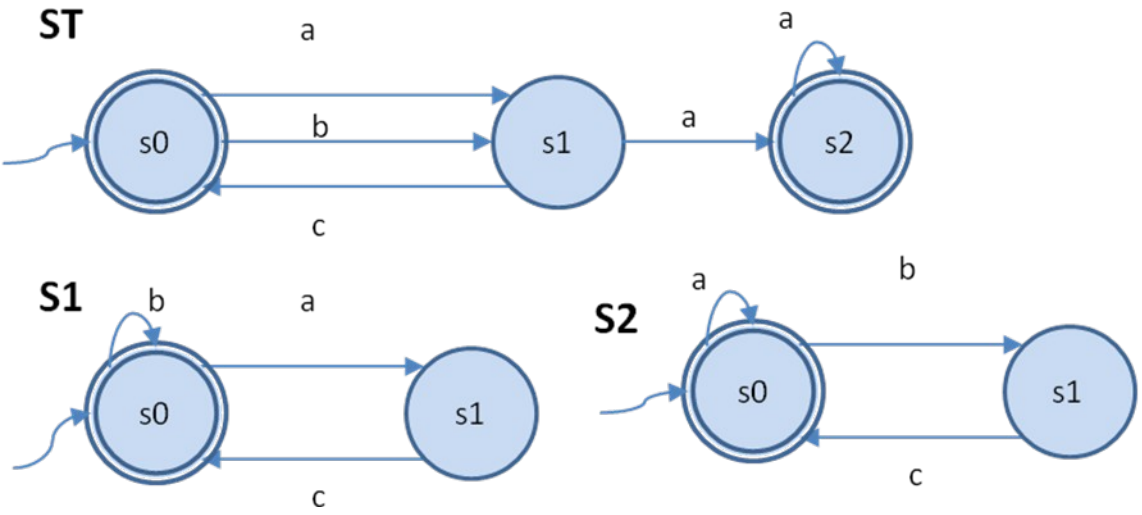

Making S1.s1 final, MCMAS answers TRUE.

```
Agent Environment
      Obsvars:
            sch : {nil,S1,ST};
      end Obsvars
      Vars:
            lastsch : {nil,S1,ST};
      end Vars
      RedStates:
      end RedStates
      Actions = {nil,S1,ST};
      Protocol:
            Other : {S1,ST};
      end Protocol
      Evolution:
            (lastsch = sch) and (sch = S1) if (Action = S1);
            (lastsch = sch) and (sch = ST) if (Action = ST);
      end Evolution
end Agent

Agent S1
      Vars:
            stateS1 : {s0,s1,s2,s_err};
            finalS1 : boolean;
            opeS1 : {a,b};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b};
      Protocol:
            Other : {a,b};
      end Protocol
      Evolution:
        (stateS1=s0) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
```

```
        (stateS1=s1) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s0) and (Action=b) and (Environment.Action=S1);
        (stateS1=s0) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s1) and (Action=a) and (Environment.Action=S1);
        (stateS1=s0) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s1) and (Action=b) and (Environment.Action=S1);
        (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 stateS1=s_err) and (Environment.Action=S1);
        (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
    end Evolution
end Agent

Agent ST
    Vars:
        stateST : {s0,s1,s_err};
        finalST : boolean;
        opeST : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
        Other : {a,b};
    end Protocol
    Evolution:
        (stateST=s1) and (finalST = false) and (opeST=b)
            if (stateST=s0) and (Action=b) and (Environment.Action=ST);
        (stateST=s1) 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=s1) and (Action=b) and (Environment.Action=ST);
        (stateST=s_err) and (opeST=a) and (finalST=false)
            if (Action=a and !(stateST=s1)) and (Environment.Action=ST);
        (stateST=s_err) and (opeST=a) and (finalST=false)
            if (Action=b and stateST=s_err) and (Environment.Action=ST);
        (stateST=stateST) and (finalST=finalST) and (opeST=opeST)
            if !(Environment.Action=ST);
    end Evolution
end Agent

Evaluation
    TargetMoved if (Environment.sch=ST);
    Replayed if ((!(Environment.sch=S1) or (S1.opeS1=ST.opeST)) and
                (ST.finalST=false or S1.finalS1=true));
    LastTargetMoved if Environment.lastsch=ST;
    Init if (Environment.lastsch=nil and Environment.sch=nil);
```

```
      Error if (S1.stateS1=s_err);
      Invalid if ST.stateST=s_err;
end Evaluation

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

Groups
      PlayersPlusEnv = { S1,Environment } ;
end Groups

Fairness
end Fairness

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

**Example 4**



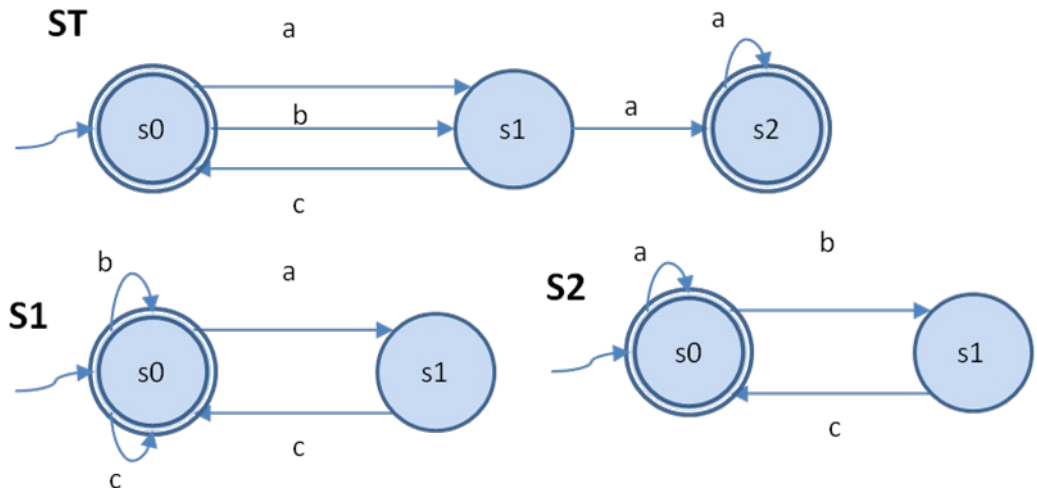

If ST performs operation a (s0 $\rightarrow_a$ s1) and the scheduler delegates it to service S2, then there's no possibility to replicate operation c (s1 $\rightarrow_c$ s0). Similarly, if the scheduler delegates operation a to S1, then ST can perform a (s1 $\rightarrow_a$ s2) but, despite it can be replicated by S2, the service S1 is left in not-final state. FALSE.

```
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};
      end Protocol
      Evolution:
            (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

Agent S1
      Vars:
            stateS1 : {s0,s1,s_err};
            finalS1 : boolean;
            opeS1 : {a,b,c};
      end Vars
      RedStates:
      end RedStates
```

```
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateS1=s1) and (finalS1 = false) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
        (stateS1=s0) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s0) and (Action=b) and (Environment.Action=S1);
        (stateS1=s0) and (finalS1 = true) and (opeS1=c)
            if (stateS1=s1) and (Action=c) and (Environment.Action=S1);
        (stateS1=s_err) and (opeS1=a) and (finalS1=false)
            if (Action=a and !(stateS1=s0)) and (Environment.Action=S1);
        (stateS1=s_err) and (opeS1=b) and (finalS1=false)
            if (Action=b and !(stateS1=s0)) and (Environment.Action=S1);
        (stateS1=s_err) and (opeS1=c) and (finalS1=false)
            if (Action=c and !(stateS1=s1)) and (Environment.Action=S1);
        (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

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

Agent ST
      Vars:
            stateST : {s0,s1,s2,s_err};
            finalST : boolean;
            opeST : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
```

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

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

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 } ;
end Groups
Fairness
end Fairness

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

**Example 4b**



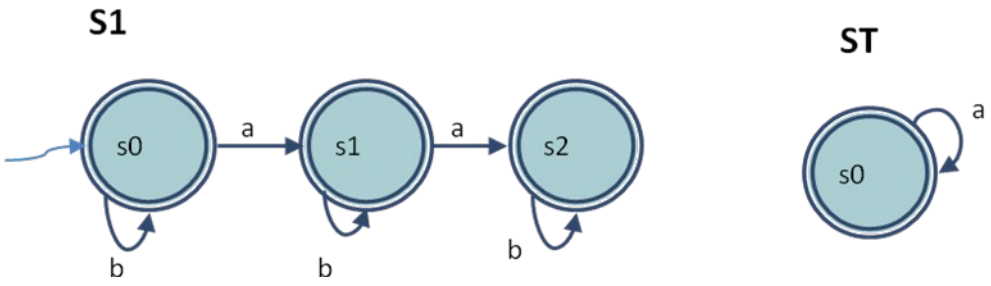

Adding a loop in S1.s0 (s0 →c s0) MCMAS answers TRUE.

```
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};
      end Protocol
      Evolution:
            (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

Agent S1
      Vars:
            stateS1 : {s0,s1,s_err};
            finalS1 : boolean;
            opeS1 : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateS1=s1) and (finalS1 = false) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
```

```
      (stateS1=s0) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s0) and (Action=b) and (Environment.Action=S1);
      (stateS1=s0) and (finalS1 = true) and (opeS1=c)
            if (stateS1=s0) and (Action=c) and (Environment.Action=S1);
      (stateS1=s0) and (finalS1 = true) and (opeS1=c)
            if (stateS1=s1) and (Action=c) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=a) and (finalS1=false)
            if (Action=a and !(stateS1=s0)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=b) and (finalS1=false)
            if (Action=b and !(stateS1=s0)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=c) and (finalS1=false)
            if (Action=c and stateS1=s_err) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

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

Agent ST
      Vars:
            stateST : {s0,s1,s2,s_err};
            finalST : boolean;
            opeST : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateST=s1) and (finalST = false) and (opeST=a)
```

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

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

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 } ;
end Groups
Fairness
end Fairness

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

**Example 5**

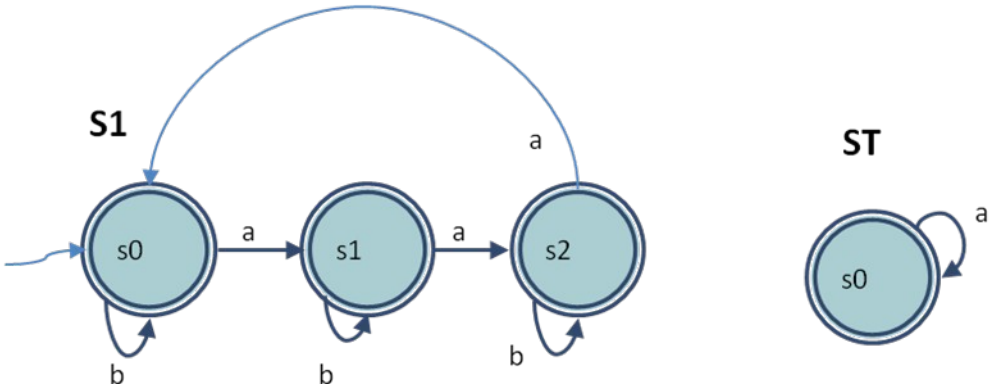



In this example, ST simply loops in s0 with operation a: but S1 can perform it just twice. MCMAS answers FALSE.

```
Agent Environment
    Obsvars:
            sch : {nil,S1,ST};
    end Obsvars
    Vars:
      lastsch : {nil,S1,ST};
    end Vars
    RedStates:
    end RedStates
    Actions = {nil,S1,ST};
    Protocol:
            Other : {S1,ST};
    end Protocol
    Evolution:
            (lastsch = sch) and (sch = S1) if (Action = S1);
            (lastsch = sch) and (sch = ST) if (Action = ST);
    end Evolution
end Agent

Agent S1
    Vars:
        stateS1 : {s0,s1,s2,s_err};
        finalS1 : boolean;
        opeS1 : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
    end Protocol
    Evolution:
      (stateS1=s1) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s0) and  (Action=a) and (Environment.Action=S1);
      (stateS1=s0) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s0) and  (Action=b) and (Environment.Action=S1);
      (stateS1=s2) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s1) and  (Action=a) and (Environment.Action=S1);
      (stateS1=s1) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s1) and  (Action=b) and (Environment.Action=S1);
      (stateS1=s2) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s2) and  (Action=b) and (Environment.Action=S1);
```

```
      (stateS1=s_err) and (opeS1=a) and (finalS1=false) if (Action=a and
            (stateS1=s2 or stateS1=s_err)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=b) and (finalS1=false)
            if (Action=b and (stateS1=s_err)) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

Agent ST
    Vars:
        stateST : {s0,s_err};
        finalST : boolean;
        opeST : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
    end Protocol
    Evolution:
      (stateST=s0) and (finalST = true) and (opeST=a)
            if (Action=a) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=a) and (finalST=false)
            if (Action=a and (stateST=s_err)) and (Environment.Action=S1);
      (stateST=s_err) and (opeST=b) and (finalST=false)
            if (Action=b) and (Environment.Action=S1);
      (stateST=stateST) and (finalST=finalST) and (opeST=opeST)
            if !(Environment.Action=ST);
      end Evolution
end Agent
Evaluation
      TargetMoved if (Environment.sch=ST);
      Replayed if ((!(Environment.sch=S1) or (S1.opeS1=ST.opeST)) and
                  (ST.finalST=false or S1.finalS1=true));
      LastTargetMoved if Environment.lastsch=ST;
      Init if (Environment.lastsch=nil and Environment.sch=nil);
      Error if (S1.stateS1=s_err);
      Invalid if ST.stateST=s_err;
end Evaluation

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

Groups
    PlayersPlusEnv = { S1,Environment } ;
end Groups
Fairness
end Fairness
Formulae
<PlayersPlusEnv> G(
  Init or Invalid or (
      (!TargetMoved -> (Replayed and !Error and LastTargetMoved))
      and
      (TargetMoved -> !LastTargetMoved)
  ));
end Formulae
```

**Example 5b**



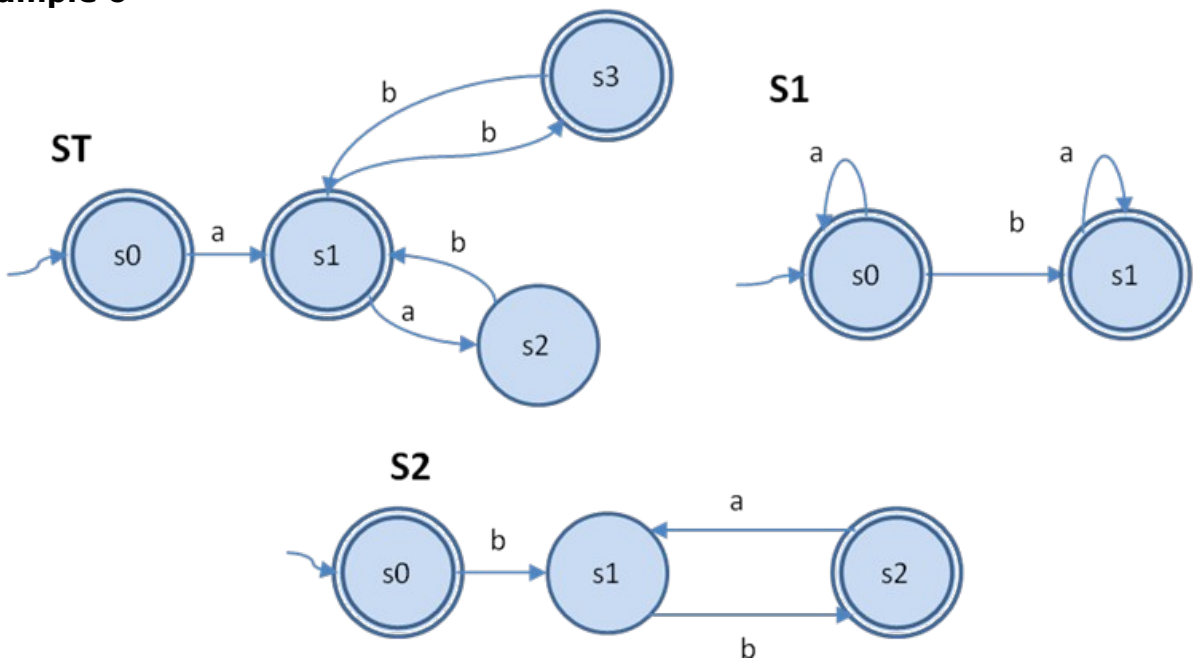

Now MCMAS answers TRUE.

```
Agent Environment
    Obsvars:
            sch : {nil,S1,ST};
    end Obsvars
    Vars:
            lastsch : {nil,S1,ST};
    end Vars
    RedStates:
    end RedStates
    Actions = {nil,S1,ST};
    Protocol:
            Other : {S1,ST};
    end Protocol
    Evolution:
            (lastsch = sch) and (sch = S1) if (Action = S1);
            (lastsch = sch) and (sch = ST) if (Action = ST);
    end Evolution
end Agent

Agent S1
    Vars:
        stateS1 : {s0,s1,s2,s_err};
        finalS1 : boolean;
        opeS1 : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
```

```
          Other : {a,b};
    end Protocol
    Evolution:
      (stateS1=s1) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s0) and  (Action=a) and (Environment.Action=S1);
      (stateS1=s0) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s0) and  (Action=b) and (Environment.Action=S1);
      (stateS1=s2) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s1) and  (Action=a) and (Environment.Action=S1);
      (stateS1=s1) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s1) and  (Action=b) and (Environment.Action=S1);
      (stateS1=s0) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s2) and  (Action=a) and (Environment.Action=S1);
      (stateS1=s2) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s2) and  (Action=b) and (Environment.Action=S1);
      (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 (stateS1=s_err)) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

Agent ST
    Vars:
        stateST : {s0,s_err};
        finalST : boolean;
        opeST : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
    end Protocol
    Evolution:
      (stateST=s0) and (finalST = true) and (opeST=a)
            if (Action=a) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=a) and (finalST=false)
            if (Action=a and (stateST=s_err)) and (Environment.Action=S1);
      (stateST=s_err) and (opeST=b) and (finalST=false)
            if (Action=b) and (Environment.Action=S1);
```

```
      (stateST=stateST) and (finalST=finalST) and (opeST=opeST)
            if !(Environment.Action=ST);
      end Evolution
end Agent

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

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

Groups
    PlayersPlusEnv = { S1,Environment } ;
end Groups

Fairness
end Fairness

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

**Example 6**



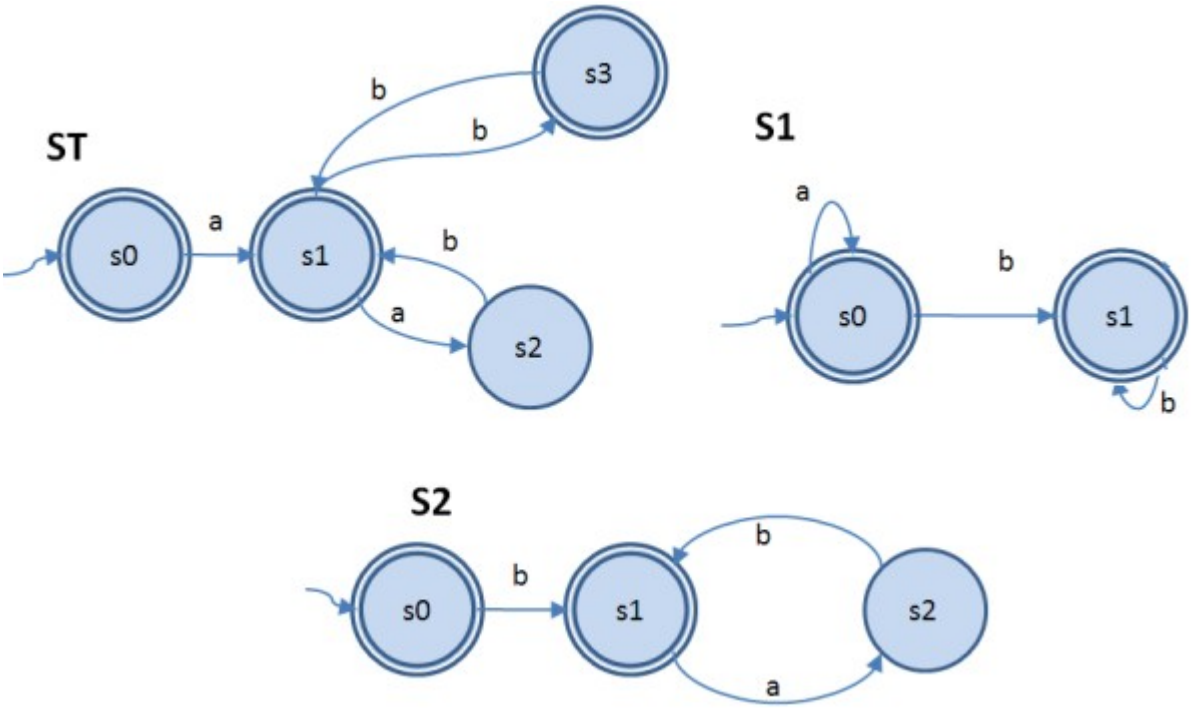

The simplest way to see way MCMAS answers FALSE is noticing that available services cannot replicate the sequence a,b,b of moves.

```
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};
    end Protocol
    Evolution:
            (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

Agent S1
    Vars:
        stateS1 : {s0,s1,s_err};
        finalS1 : boolean;
        opeS1 : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
```

```
    end Protocol
    Evolution:
      (stateS1=s0) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
      (stateS1=s1) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s0) and (Action=b) and (Environment.Action=S1);
      (stateS1=s1) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s1) and (Action=a) and (Environment.Action=S1);
      (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 !(stateS1=s0)) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

Agent S2
    Vars:
        stateS2 : {s0,s1,s2,s_err};
        finalS2 : boolean;
        opeS2 : {a,b};
    end Vars
    RedStates:
    end RedStates
    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=s2) and (finalS2 = true)  and (opeS2=b)
            if (stateS2=s1) and  (Action=b) and (Environment.Action=S2);
      (stateS2=s1) and (finalS2 = false)  and (opeS2=a)
            if (stateS2=s2) and  (Action=a) and (Environment.Action=S2);
      (stateS2=s_err) and (opeS2=a) and (finalS2=false)
            if (Action=a and !(stateS2=s2)) 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

Agent ST
    Vars:
        stateST : {s0,s1,s2,s3,s_err};
        finalST : boolean;
        opeST : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
    end Protocol
    Evolution:
      (stateST=s1) and (finalST = true) 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=s3) and (finalST = true) and (opeST=b)
            if (stateST=s1) and  (Action=b) and (Environment.Action=ST);
      (stateST=s1) and (finalST = true) and (opeST=b)
            if  (stateST=s2) and (Action=b) and (Environment.Action=ST);
      (stateST=s1) and (finalST = true) and (opeST=b)
            if  (stateST=s3) and (Action=b) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=a) and (finalST=false) if (Action=a and
            !(stateST=s0 or stateST=s1)) and (Environment.Action=S1);
      (stateST=s_err) and (opeST=b) and (finalST=false) if (Action=b and
            (stateST=s0 or  stateST=s_err)) and (Environment.Action=S1);
      (stateST=stateST) and (finalST=finalST) and (opeST=opeST)
            if !(Environment.Action=ST);
      end Evolution
end Agent

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

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 } ;
end Groups

Fairness
end Fairness

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

**Example 6b**



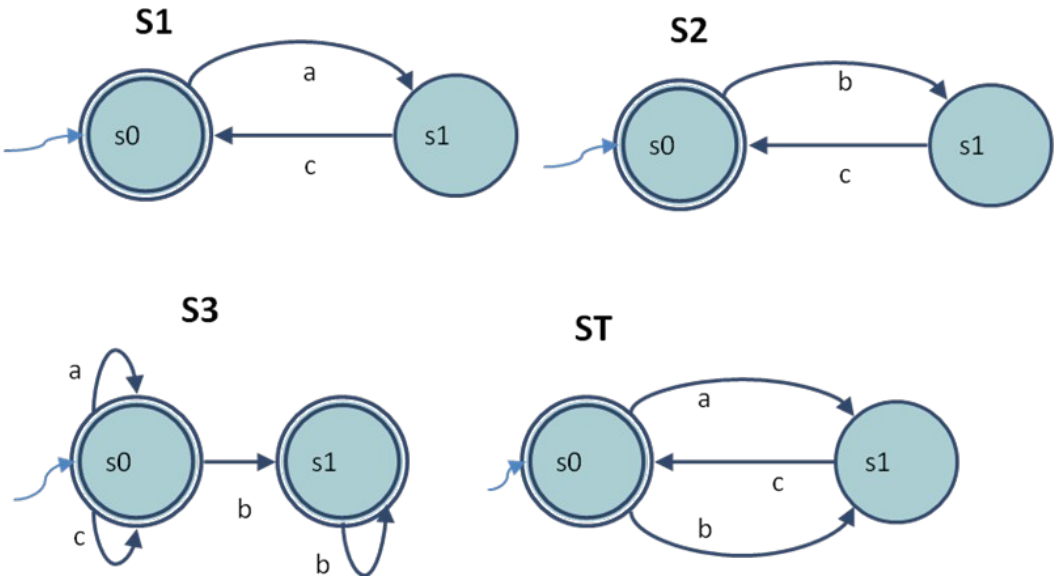

MCMAS's result is: TRUE

```
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};
    end Protocol
    Evolution:
            (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

Agent S1
    Vars:
        stateS1 : {s0,s1,s_err};
        finalS1 : boolean;
        opeS1 : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
```

```
    Protocol:
          Other : {a,b};
    end Protocol
    Evolution:
      (stateS1=s0) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
      (stateS1=s1) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s0) and (Action=b) and (Environment.Action=S1);
      (stateS1=s1) and (finalS1 = true) and (opeS1=b)
            if (stateS1=s1) and (Action=b) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=a) and (finalS1=false)
            if (Action=a and !(stateS1=s0)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=b) and (finalS1=false)
            if (Action=b and stateS1=s_err) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

Agent S2
    Vars:
        stateS2 : {s0,s1,s2,s_err};
        finalS2 : boolean;
        opeS2 : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
    end Protocol
    Evolution:
      (stateS2=s1) and (finalS2 = true) and (opeS2=b)
            if (stateS2=s0) and  (Action=b) and (Environment.Action=S2);
      (stateS2=s2) and (finalS2 = false)  and (opeS2=a)
            if (stateS2=s1) and  (Action=a) and (Environment.Action=S2);
      (stateS2=s1) and (finalS2 = true)  and (opeS2=b)
            if (stateS2=s2) and  (Action=b) and (Environment.Action=S2);
      (stateS2=s_err) and (opeS2=a) and (finalS2=false)
            if (Action=a and !(stateS2=s1)) and (Environment.Action=S2);
      (stateS2=s_err) and (opeS2=b) and (finalS2=false) if (Action=b and
            (stateS2=s1 or stateS2=s_err)) and (Environment.Action=S2);
      (stateS2=stateS2) and (finalS2=finalS2) if !(Environment.Action=S2);
      end Evolution
end Agent

Agent ST
    Vars:
        stateST : {s0,s1,s2,s3,s_err};
        finalST : boolean;
        opeST : {a,b};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b};
    Protocol:
          Other : {a,b};
    end Protocol
    Evolution:
      (stateST=s1) and (finalST = true) 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=s3) and (finalST = true) and (opeST=b)
            if (stateST=s1) and  (Action=b) and (Environment.Action=ST);
      (stateST=s1) and (finalST = true) and (opeST=b)
            if  (stateST=s2) and (Action=b) and (Environment.Action=ST);
      (stateST=s1) and (finalST = true) and (opeST=b)
            if  (stateST=s3) and (Action=b) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=a) and (finalST=false) if (Action=a and
            !(stateST=s0 or stateST=s1)) and (Environment.Action=ST);
      (stateST=s_err) and (opeST=b) and (finalST=false) if (Action=b
            and (stateST=s0 or stateST=s_err)) and (Environment.Action=ST);
      (stateST=stateST) and (finalST=finalST) and (opeST=opeST)
            if !(Environment.Action=ST);
      end Evolution
end Agent

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

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 } ;
end Groups

Fairness
end Fairness

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

**Example 7**




S1 and S2 can easily exploited to mimic ST behavior. MCMAS result: TRUE

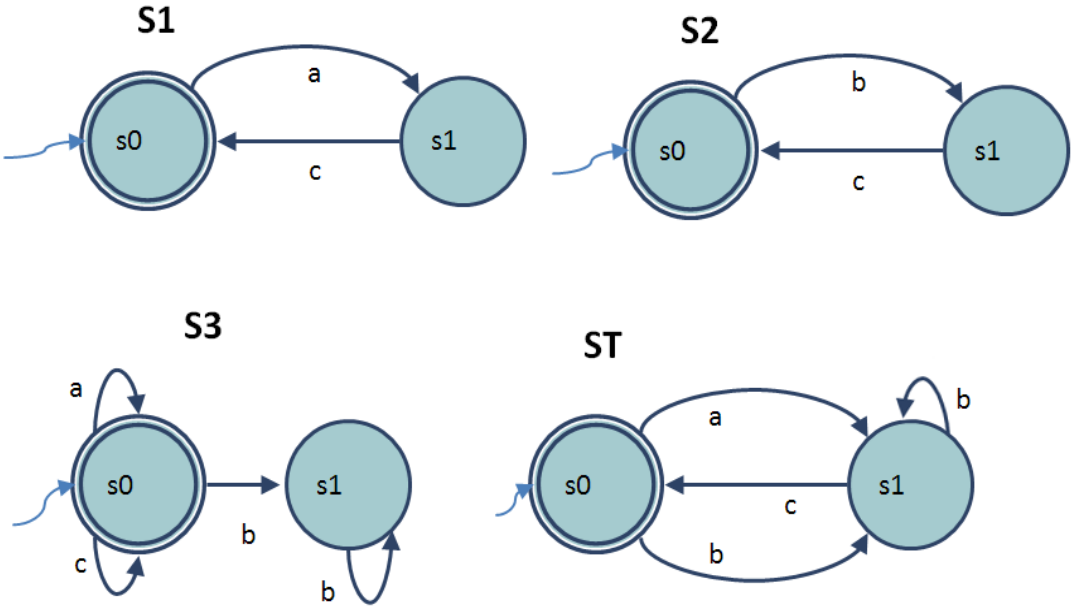
```
Agent Environment
    Obsvars:
            sch : {nil,S1,S2,S3,ST};
    end Obsvars
    Vars:
      lastsch : {nil,S1,S2,S3,ST};
    end Vars
    RedStates:
    end RedStates
    Actions = {nil,S1,S2,S3,ST};
    Protocol:
            Other : {S1,S2,S3,ST};
    end Protocol
    Evolution:
            (lastsch = sch) and (sch = S1) if (Action = S1);
            (lastsch = sch) and (sch = S2) if (Action = S2);
            (lastsch = sch) and (sch = S3) if (Action = S3);
            (lastsch = sch) and (sch = ST) if (Action = ST);
    end Evolution
end Agent

Agent S1
    Vars:
        stateS1 : {s0,s1,s_err};
        finalS1 : boolean;
        opeS1 : {a,b,c};
    end Vars
```

```
    RedStates:
    end RedStates
    Actions = {a,b,c};
    Protocol:
          Other : {a,b,c};
    end Protocol
    Evolution:
      (stateS1=s1) and (finalS1 = false) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
      (stateS1=s0) and (finalS1 = true)  and (opeS1=c)
            if (stateS1=s1) and (Action=c) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=a) and (finalS1=false)
            if (Action=a and !(stateS1=s0)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=c) and (finalS1=false)
            if (Action=c and !(stateS1=s1)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=b) if (Action=b) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

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

Agent S3
```

```
    Vars:
        stateS3 : {s0,s1,s_err};
        finalS3 : boolean;
        opeS3 : {a,b,c};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b,c};
    Protocol:
          Other : {a,b,c};
    end Protocol
    Evolution:
      (stateS3=s0) and (finalS3 = true) and (opeS3=a)
            if (stateS3=s0) and (Action=a) and (Environment.Action=S3);
      (stateS3=s0) and (finalS3 = true) and (opeS3=c)
            if (stateS3=s0) and  (Action=c) and (Environment.Action=S3);
      (stateS3=s1) and (finalS3 = true) and (opeS3=b)
            if (stateS3=s0) and  (Action=b) and (Environment.Action=S3);
      (stateS3=s1) and (finalS3 = true) and (opeS3=b)
            if (stateS3=s1) and  (Action=b) and (Environment.Action=S3);
      (stateS3=s_err) and (opeS3=a) and (finalS3=false)
            if (Action=a and !(stateS3=s0)) and (Environment.Action=S3);
      (stateS3=s_err) and (opeS3=b) and (finalS3=false)
            if (Action=b and stateS3=s_err) and (Environment.Action=S3);
      (stateS3=s_err) and (opeS3=c) and (finalS3=false)
            if (Action=c and !(stateS3=s0)) and (Environment.Action=S3);
      (stateS3=stateS3) and (finalS3=finalS3) if !(Environment.Action=S3);
      end Evolution
end Agent

Agent ST
    Vars:
        stateST : {s0,s1,s_err};
        finalST : boolean;
        opeST : {a,b,c};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b,c};
    Protocol:
          Other : {a,b,c};
    end Protocol
    Evolution:
      (stateST=s1) and (finalST = false) and (opeST=a)
            if (stateST=s0) and (Action=a) and (Environment.Action=ST);
      (stateST=s1) and (finalST = false) and (opeST=b)
            if (stateST=s0) and (Action=b) and (Environment.Action=ST);
```

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

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
                  (!(Environment.sch=S3) or (S3.opeS3=ST.opeST)))
                  and (ST.finalST=false or (S1.finalS1=true and S2.finalS2=true
                  and S3.finalS3=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) or (S3.stateS3=s_err);
      Invalid if ST.stateST=s_err;
end Evaluation

InitStates
      (S1.stateS1=s0) and (S2.stateS2=s0) and (S3.stateS3=s0) and
      (ST.stateST=s0) and (Environment.sch=nil) and
      (S1.finalS1=true) and (S2.finalS2=true) and (S3.finalS3=true) and
      (ST.finalST=true) and (Environment.lastsch=nil);
end InitStates

Groups
    PlayersPlusEnv = { S1,S2,S3,Environment } ;
end Groups
Fairness
end Fairness

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

**Example 7b**




The loop added in ST.s1 involves using S3, but S3.s1 is not final. Result: FALSE

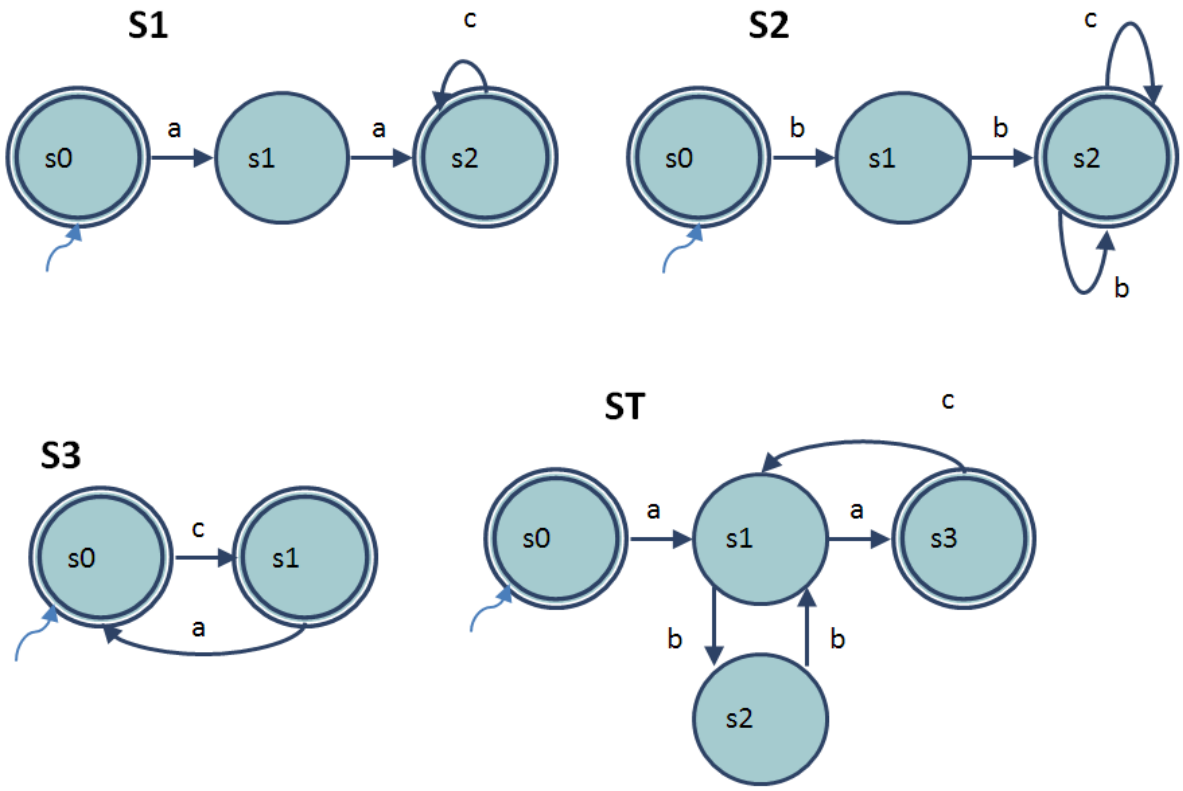
```
Agent Environment
    Obsvars:
            sch : {nil,S1,S2,S3,ST};
    end Obsvars
    Vars:
      lastsch : {nil,S1,S2,S3,ST};
    end Vars
    RedStates:
    end RedStates
    Actions = {nil,S1,S2,S3,ST};
    Protocol:
            Other : {S1,S2,S3,ST};
    end Protocol
    Evolution:
            (lastsch = sch) and (sch = S1) if (Action = S1);
            (lastsch = sch) and (sch = S2) if (Action = S2);
            (lastsch = sch) and (sch = S3) if (Action = S3);
            (lastsch = sch) and (sch = ST) if (Action = ST);
    end Evolution
end Agent

Agent S1
    Vars:
        stateS1 : {s0,s1,s_err};
        finalS1 : boolean;
        opeS1 : {a,b,c};
    end Vars
    RedStates:
    end RedStates
```

```
    Actions = {a,b,c};
    Protocol:
          Other : {a,b,c};
    end Protocol
    Evolution:
      (stateS1=s1) and (finalS1 = false) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
      (stateS1=s0) and (finalS1 = true)  and (opeS1=c)
            if (stateS1=s1) and (Action=c) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=a) and (finalS1=false)
            if (Action=a and !(stateS1=s0)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=c) and (finalS1=false)
            if (Action=c and !(stateS1=s1)) and (Environment.Action=S1);
      (stateS1=s_err) and (opeS1=b) and (finalS1=false)
            if (Action=b) and (Environment.Action=S1);
      (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

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

Agent S3
    Vars:
        stateS3 : {s0,s1,s_err};
```

```
        finalS3 : boolean;
        opeS3 : {a,b,c};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b,c};
    Protocol:
          Other : {a,b,c};
    end Protocol
    Evolution:
      (stateS3=s0) and (finalS3 = true) and (opeS3=a)
            if (stateS3=s0) and (Action=a) and (Environment.Action=S3);
      (stateS3=s0) and (finalS3 = true) and (opeS3=c)
            if (stateS3=s0) and  (Action=c) and (Environment.Action=S3);
      (stateS3=s1) and (finalS3 = false) and (opeS3=b)
            if (stateS3=s0) and  (Action=b) and (Environment.Action=S3);
      (stateS3=s1) and (finalS3 = false) and (opeS3=b)
            if (stateS3=s1) and  (Action=b) and (Environment.Action=S3);
      (stateS3=s_err) and (opeS3=a) and (finalS3=false)
            if (Action=a and !(stateS3=s0)) and (Environment.Action=S3);
      (stateS3=s_err) and (opeS3=b) and (finalS3=false)
            if (Action=b and stateS3=s_err) and (Environment.Action=S3);
      (stateS3=s_err) and (opeS3=c) and (finalS3=false)
            if (Action=c and !(stateS3=s0)) and (Environment.Action=S3);
      (stateS3=stateS3) and (finalS3=finalS3) if !(Environment.Action=S3);
      end Evolution
end Agent

Agent ST
    Vars:
        stateST : {s0,s1,s_err};
        finalST : boolean;
        opeST : {a,b,c};
    end Vars
    RedStates:
    end RedStates
    Actions = {a,b,c};
    Protocol:
          Other : {a,b,c};
    end Protocol
    Evolution:
      (stateST=s1) and (finalST = false) and (opeST=a)
            if (stateST=s0) and (Action=a) and (Environment.Action=ST);
      (stateST=s1) and (finalST = false) and (opeST=b)
            if (stateST=s0) and (Action=b) and (Environment.Action=ST);
      (stateST=s1) and (finalST = false) and (opeST=b)
            if (stateST=s1) and (Action=b) and (Environment.Action=ST);
      (stateST=s0) and (finalST = true) and (opeST=c)
            if (stateST=s1) and (Action=c) and (Environment.Action=ST);
```

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

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
                  (!(Environment.sch=S3) or (S3.opeS3=ST.opeST)))
                  and (ST.finalST=false or (S1.finalS1=true and S2.finalS2=true
                  and S3.finalS3=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) or (S3.stateS3=s_err);
      Invalid if ST.stateST=s_err;
end Evaluation

InitStates
      (S1.stateS1=s0) and (S2.stateS2=s0) and (S3.stateS3=s0) and
      (ST.stateST=s0) and (Environment.sch=nil) and
      (S1.finalS1=true) and (S2.finalS2=true) and (S3.finalS3=true) and
      (ST.finalST=true) and (Environment.lastsch=nil);
end InitStates

Groups
    PlayersPlusEnv = { S1,S2,S3,Environment } ;
end Groups

Fairness
end Fairness

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

**Example 8**




MCMAS result: TRUE

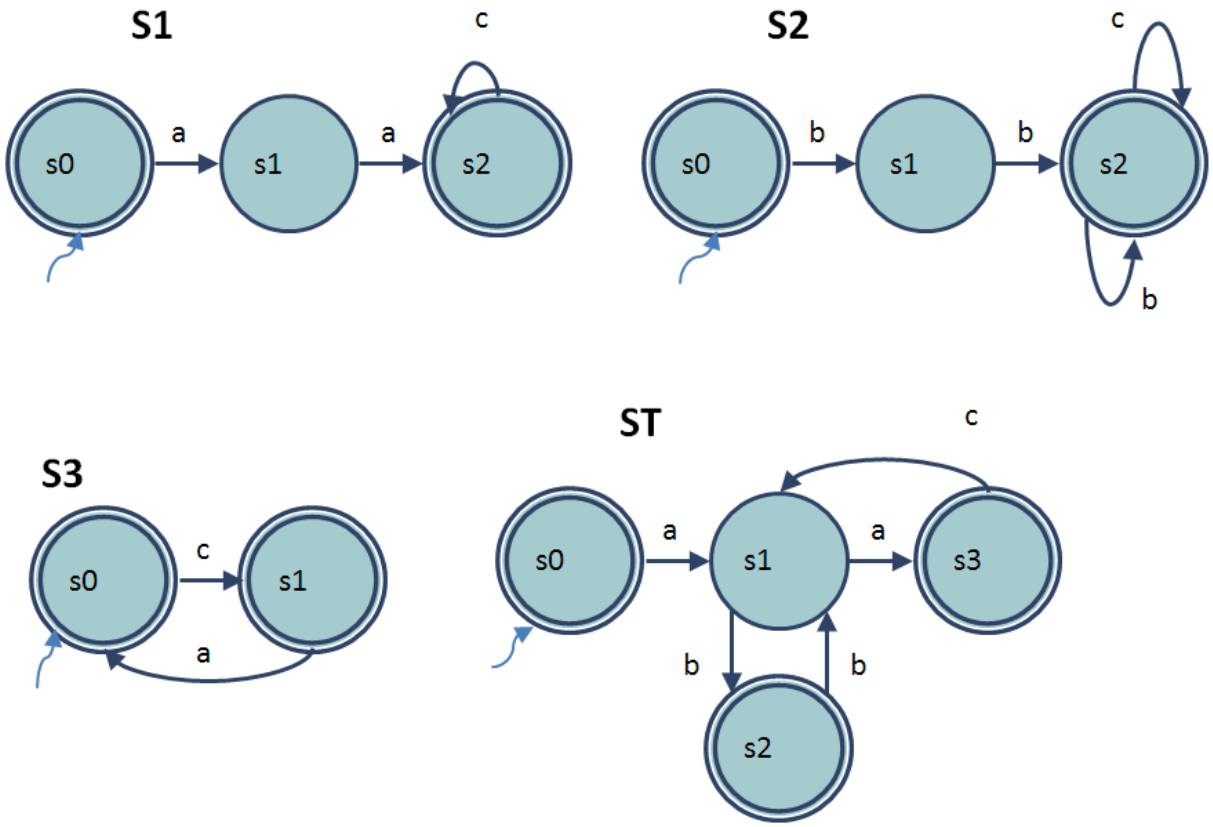
```
Agent Environment
      Obsvars:
            sch : {nil,S1,S2,S3,ST};
      end Obsvars
      Vars:
            lastsch : {nil,S1,S2,S3,ST};
      end Vars
      RedStates:
      end RedStates
      Actions = {nil,S1,S2,S3,ST};
      Protocol:
            Other : {S1,S2,S3,ST};
      end Protocol
      Evolution:
            (lastsch = sch) and (sch = S1) if (Action = S1);
            (lastsch = sch) and (sch = S2) if (Action = S2);
            (lastsch = sch) and (sch = S3) if (Action = S3);
            (lastsch = sch) and (sch = ST) if (Action = ST);
      end Evolution
end Agent

Agent S1
      Vars:
            stateS1 : {s0,s1,s2,s_err};
            finalS1 : boolean;
```

```
            opeS1 : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateS1=s1) and (finalS1 = false) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
        (stateS1=s2) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s1) and (Action=a) and (Environment.Action=S1);
        (stateS1=s2) and (finalS1 = true) and (opeS1=c)
            if (stateS1=s2) and (Action=c) and (Environment.Action=S1);
        (stateS1=s_err) and (opeS1=a) and (finalS1=false) if (Action=a
            and (stateS1=s2 or stateS1=s_err)) and (Environment.Action=S1);
        (stateS1=s_err) and (opeS1=b) and (finalS1=false)
            if (Action=b) and (Environment.Action=S1);
        (stateS1=s_err) and (opeS1=c) and (finalS1=false)
            if (Action=c and !(stateS1=s2)) and (Environment.Action=S1);
        (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

Agent S2
      Vars:
            stateS2 : {s0,s1,s2,s_err};
            finalS2 : boolean;
            opeS2 : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateS2=s1) and (finalS2 = false) and (opeS2=b)
            if (stateS2=s0) and (Action=b) and (Environment.Action=S2);
        (stateS2=s2) and (finalS2 = true) and (opeS2=b)
            if (stateS2=s1) and (Action=b) and (Environment.Action=S2);
        (stateS2=s2) and (finalS2 = true) and (opeS2=b)
            if (stateS2=s2) and (Action=b) and (Environment.Action=S2);
        (stateS2=s2) and (finalS2 = true) and (opeS2=c)
            if (stateS2=s2) and (Action=c) and (Environment.Action=S2);
        (stateS2=s_err) and (opeS2=a) and (finalS2=false)
            if (Action=a) 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=s_err) and (opeS2=c) and (finalS2=false)
            if (Action=c and !(stateS2=s2)) and (Environment.Action=S2);
      (stateS2=stateS2) and (finalS2=finalS2) if !(Environment.Action=S2);
      end Evolution
end Agent

Agent S3
      Vars:
            stateS3 : {s0,s1,s_err};
```

```
            finalS3 : boolean;
            opeS3 : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateS3=s1) and (finalS3 = true) and (opeS3=c)
            if (stateS3=s0) and (Action=c) and (Environment.Action=S3);
        (stateS3=s0) and (finalS3 = true) and (opeS3=a)
            if (stateS3=s1) and (Action=a) and (Environment.Action=S3);
        (stateS3=s_err) and (opeS3=a) and (finalS3=false)
            if (Action=a and !(stateS3=s1)) and (Environment.Action=S3);
        (stateS3=s_err) and (opeS3=b) and (finalS3=false)
            if (Action=b) and (Environment.Action=S3);
        (stateS3=s_err) and (opeS3=c) and (finalS3=false)
            if (Action=c and !(stateS3=s0)) and (Environment.Action=S3);
        (stateS3=stateS3) and (finalS3=finalS3) if !(Environment.Action=S3);
      end Evolution
end Agent

Agent ST
      Vars:
            stateST : {s0,s1,s2,s3,s_err};
            finalST : boolean;
            opeST : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateST=s1) and (finalST = false) and (opeST=a)
            if (stateST=s0) and (Action=a) and (Environment.Action=ST);
        (stateST=s3) and (finalST = true) and  (opeST=a)
            if (stateST=s1) and (Action=a) and (Environment.Action=ST);
        (stateST=s2) and (finalST = false) and (opeST=b)
            if (stateST=s1) and (Action=b) and (Environment.Action=ST);
        (stateST=s1) and (finalST = false) and (opeST=b)
            if (stateST=s2) and (Action=b) and (Environment.Action=ST);
        (stateST=s1) and (finalST = true) and  (opeST=c)
            if (stateST=s3) and (Action=c) and (Environment.Action=ST);
        (stateST=s_err) and (opeST=a) and (finalS3=false) if (Action=a
            and (stateST=s3 or stateST=s_err)) and (Environment.Action=ST);
        (stateST=s_err) and (opeST=b) and (finalS3=false) if (Action=b
            and !(stateST=s1 or stateST=s2)) and (Environment.Action=ST);
        (stateST=s_err) and (opeST=c) and (finalS3=false)
            if (Action=c and !(stateST=s3)) and (Environment.Action=ST);
        (stateST=stateST) and (finalST=finalST) and (opeST=opeST)
            if !(Environment.Action=ST);
      end Evolution
end Agent

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 (!
                  (Environment.sch=S3) or (S3.opeS3=ST.opeST)))
                  and (ST.finalST=false or (S1.finalS1=true and S2.finalS2=true
                  and S3.finalS3=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) or (S3.stateS3=s_err);
      Invalid if ST.stateST=s_err;
end Evaluation

InitStates
      (S1.stateS1=s0) and (S2.stateS2=s0) and (S3.stateS3=s0) and
      (ST.stateST=s0) and (Environment.sch=nil) and
      (S1.finalS1=true) and (S2.finalS2=true) and (S3.finalS3=true) and
      (ST.finalST=true) and (Environment.lastsch=nil);
end InitStates

Groups
      PlayersPlusEnv = { S1,S2,S3,Environment } ;
end Groups

Fairness
end Fairness

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

**Example 8b**




State ST.s2 is now final. MCMAS result: FALSE

```
Agent Environment
      Obsvars:
            sch : {nil,S1,S2,S3,ST};
      end Obsvars
      Vars:
            lastsch : {nil,S1,S2,S3,ST};
      end Vars
      RedStates:
      end RedStates
      Actions = {nil,S1,S2,S3,ST};
      Protocol:
            Other : {S1,S2,S3,ST};
      end Protocol
      Evolution:
            (lastsch = sch) and (sch = S1) if (Action = S1);
            (lastsch = sch) and (sch = S2) if (Action = S2);
            (lastsch = sch) and (sch = S3) if (Action = S3);
            (lastsch = sch) and (sch = ST) if (Action = ST);
      end Evolution
end Agent

Agent S1
```

```
      Vars:
            stateS1 : {s0,s1,s2,s_err};
            finalS1 : boolean;
            opeS1 : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateS1=s1) and (finalS1 = false) and (opeS1=a)
            if (stateS1=s0) and (Action=a) and (Environment.Action=S1);
        (stateS1=s2) and (finalS1 = true) and (opeS1=a)
            if (stateS1=s1) and (Action=a) and (Environment.Action=S1);
        (stateS1=s2) and (finalS1 = true) and (opeS1=c)
            if (stateS1=s2) and (Action=c) and (Environment.Action=S1);
        (stateS1=s_err) and (opeS1=a) and (finalS1=false) if (Action=a
            and (stateS1=s2 or stateS1=s_err)) and (Environment.Action=S1);
        (stateS1=s_err) and (opeS1=b) and (finalS1=false)
            if (Action=b) and (Environment.Action=S1);
        (stateS1=s_err) and (opeS1=c) and (finalS1=false)
            if (Action=c and !(stateS1=s2)) and (Environment.Action=S1);
        (stateS1=stateS1) and (finalS1=finalS1) if !(Environment.Action=S1);
      end Evolution
end Agent

Agent S2
      Vars:
            stateS2 : {s0,s1,s2,s_err};
            finalS2 : boolean;
            opeS2 : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateS2=s1) and (finalS2 = false) and (opeS2=b)
            if (stateS2=s0) and (Action=b) and (Environment.Action=S2);
        (stateS2=s2) and (finalS2 = true) and (opeS2=b)
            if (stateS2=s1) and (Action=b) and (Environment.Action=S2);
        (stateS2=s2) and (finalS2 = true) and (opeS2=b)
            if (stateS2=s2) and (Action=b) and (Environment.Action=S2);
        (stateS2=s2) and (finalS2 = true) and (opeS2=c)
            if (stateS2=s2) and (Action=c) and (Environment.Action=S2);
        (stateS2=s_err) and (opeS2=a) and (finalS2=false)
            if (Action=a) 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=s_err) and (opeS2=c) and (finalS2=false)
            if (Action=c and !(stateS2=s2)) and (Environment.Action=S2);
      (stateS2=stateS2) and (finalS2=finalS2) if !(Environment.Action=S2);
      end Evolution
end Agent
```

```
Agent S3
      Vars:
            stateS3 : {s0,s1,s_err};
            finalS3 : boolean;
            opeS3 : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateS3=s1) and (finalS3 = true) and (opeS3=c)
            if (stateS3=s0) and (Action=c) and (Environment.Action=S3);
        (stateS3=s0) and (finalS3 = true) and (opeS3=a)
            if (stateS3=s1) and (Action=a) and (Environment.Action=S3);
        (stateS3=s_err) and (opeS3=a) and (finalS3=false)
            if (Action=a and !(stateS3=s1)) and (Environment.Action=S3);
        (stateS3=s_err) and (opeS3=b) and (finalS3=false)
            if (Action=b) and (Environment.Action=S3);
        (stateS3=s_err) and (opeS3=c) and (finalS3=false)
            if (Action=c and !(stateS3=s0)) and (Environment.Action=S3);
        (stateS3=stateS3) and (finalS3=finalS3) if !(Environment.Action=S3);
      end Evolution
end Agent

Agent ST
      Vars:
            stateST : {s0,s1,s2,s3,s_err};
            finalST : boolean;
            opeST : {a,b,c};
      end Vars
      RedStates:
      end RedStates
      Actions = {a,b,c};
      Protocol:
            Other : {a,b,c};
      end Protocol
      Evolution:
        (stateST=s1) and (finalST = false) and (opeST=a)
            if (stateST=s0) and (Action=a) and (Environment.Action=ST);
        (stateST=s3) and (finalST = true) and  (opeST=a)
            if (stateST=s1) and (Action=a) and (Environment.Action=ST);
        (stateST=s2) and (finalST = true) and (opeST=b)
            if (stateST=s1) and (Action=b) and (Environment.Action=ST);
        (stateST=s1) and (finalST = false) and (opeST=b)
            if (stateST=s2) and (Action=b) and (Environment.Action=ST);
        (stateST=s1) and (finalST = true) and  (opeST=c)
            if (stateST=s3) and (Action=c) and (Environment.Action=ST);
        (stateST=s_err) and (opeST=a) and (finalST=false) if (Action=a
            and (stateST=s3 or stateST=s_err)) and (Environment.Action=ST);
        (stateST=s_err) and (opeST=b) and (finalST=false) if (Action=b
            and !(stateST=s1 or  stateST=s2)) and (Environment.Action=ST);
        (stateST=s_err) and (opeST=c) and (finalST=false)
            if (Action=c and !(stateST=s3)) and (Environment.Action=ST);
        (stateST=stateST) and (finalST=finalST) and (opeST=opeST)
            if !(Environment.Action=ST);
      end Evolution
```

```
end Agent

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 (!
                  (Environment.sch=S3) or (S3.opeS3=ST.opeST)))
                  and (ST.finalST=false or (S1.finalS1=true and S2.finalS2=true
                  and S3.finalS3=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) or (S3.stateS3=s_err);
      Invalid if ST.stateST=s_err;
end Evaluation

InitStates
      (S1.stateS1=s0) and (S2.stateS2=s0) and (S3.stateS3=s0) and
      (ST.stateST=s0) and (Environment.sch=nil) and
      (S1.finalS1=true) and (S2.finalS2=true) and (S3.finalS3=true) and
      (ST.finalST=true) and (Environment.lastsch=nil);
end InitStates

Groups
      PlayersPlusEnv = { S1,S2,S3,Environment } ;
end Groups

Fairness
end Fairness

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