

# Considerations about the implementation of an ATL model checker

Laura Florentina Stoica, Florin Stoica

## Abstract

The problem of model checking is to verify if a finite-state abstraction of a reactive system satisfies a temporal-logic specification. A Computation Tree Logic (CTL) specification is interpreted over Kripke structures, which provide a model for the computations of a closed system (the behaviour is completely determined by the state of the system). In order to capture compositions of open systems, we present an extension of CTL, the alternating-time temporal logic (ATL), which is interpreted over game structures. In this paper we will show how our original ANTLR-based model checker for CTL can be modified to check an ATL specification, using a data structure suitable for multi-graph representation of a concurrent game structure.

## 1 Introduction

Model checking is a technology often used for the automated system verification. The model checking algorithms are currently used as verification techniques implemented in varied programming environments. The verified system can be a physical system or a real-time concurrent program. The behavior of a closed system can be described by the Kripke model. The Kripke models are based on the states and use the SMV (Symbolic Model Verifier) technique. The SMV model checking takes as input the model and formula then check whether or not the formula is satisfied or not by the model. [1]

A Computation Tree Logic (CTL) model is defined as a directed graph and its semantics is interpreted over a Kripke structure. A Kripke model  $M$  over a set of atomic propositions, denoted by AP, is a triple  $M = \langle S, Rel, P: S \rightarrow 2^{AP} \rangle$  where  $S$  is a finite set of states,  $Rel \subseteq S \times S$  is a transition relation, and  $P: S \rightarrow 2^{AP}$  is a function that assigns each state with a set of atomic propositions from AP.

CTL model checker is branching-time logic, meaning that its formulas are interpreted over all paths beginning in a given state of the Kripke structure.

For each state from graph  $M$  there is a successor and a path composed by a sequence of some states.

Details about formal definitions for syntax and semantics of a CTL model checker can be found in paper [3].

A Kripke structure offers a natural model for the computations of a *closed system*, whose behavior is completely determined by the state of the system. The compositional modeling and design of reactive systems requires each component to be viewed as an *open system*.

An **open system** is a system that interacts with its environment and whose behaviour depends on the state of the system as well as the behaviour of the environment. In order to construct models suitable for open systems, was defined the Alternating-time Temporal Logic (ATL) [5].

ATL extends CTL by replacing the path quantifier's  $\forall$  and  $\exists$  by cooperation modalities  $\langle\langle A \rangle\rangle$ , where A is a team of agents. A formula  $\langle\langle A \rangle\rangle \varphi$  expresses that the team A has a collective strategy to enforce  $\varphi$ .

ATL is a branching-time temporal logic that naturally describes computations of multi-agent system and multiplayer games. It offers selective quantification over program-paths that are possible outcomes of games [5]. ATL uses alternating-time formulas to construct model-checkers in order to address problems such as receptiveness, realizability, and controllability.

Over structures without fairness constraints, the model-checking complexity of ATL is linear in the size of the game structure and length of the formula, and the symbolic model-checking algorithm for CTL extends with few modifications to ATL.

From a formal point of view, implementation of a CTL model checker will be equated with implementation of an algebraic compiler which can be defined using  $\Sigma$ -algebras and  $\Sigma$ -heterogenic homomorphism as  $C:L_s \rightarrow L_t$ , where  $L_s$  is the source language and  $L_t$  is the target language. The source language  $L_s$  is CTL, and the target language  $L_t$  is a language which describes the set of nodes from the model  $M$  where a CTL formulas  $f$  is satisfied. The algebraic compiler  $C$  translates formula  $f$  of the CTL model to set of nodes  $S'$  over which formula  $f$  is satisfied. That is,  $C(f) = S'$  where  $S' = \{s \in S \mid (M, s) \models f\}$ .

The CTL language is defined as a  $\Sigma$ -language [2]. The operator scheme  $\Sigma_{ctl}$  is defined as a triple  $\langle S_{ctl}, O_{ctl}, \sigma_{ctl} \rangle$  where set  $S_{ctl}$  contains the representations of the CTL formulas,  $O_{ctl} = \{T, \perp, \neg, \wedge, \vee, \rightarrow, AX, EX, AU, EU, EF, AF, EG, AG\}$  is the set of operators, and the  $\sigma_{ctl}: O_{ctl} \rightarrow S_{ctl}^* \times S_{ctl}$  is a function which defines the signature of the operators [2]. The CTL model checker can be defined as the  $\Sigma_{ctl}$ -language given in the form  $L_{ctl} = \langle Sem_{ctl}, Syn_{ctl}, L_{ctl}: Sem_{ctl} \rightarrow Syn_{ctl} \rangle$  where  $Syn_{ctl}$  is the word algebra of the operator scheme  $\Sigma_{ctl}$  generated by the operations from  $O_{ctl}$  and a finite set of variables, representing atomic propositions, denoted by  $AP$ .  $Sem_{ctl}$  represents CTL semantic algebra defined over the set of CTL formulas which are satisfied by the CTL model  $M$ .  $L_{ctl}$  is a mapping which associates the set of satisfied formulas from  $Sem_{ctl}$  to CTL expressions from  $Syn_{ctl}$  which satisfy these formulas. [4]

In the same way we can define the ATL language. The operator scheme  $\Sigma_{atl}$  is defined as a triple  $\langle S_{atl}, O_{atl}, \sigma_{atl} \rangle$  where set  $S_{atl}$  contains the representations of the ATL formulas,  $O_{atl} = \{\neg, \vee, \wedge, \rightarrow, \diamond, \circ, \square, U\}$  is the set of operators, and the  $\sigma_{atl}: O_{atl} \rightarrow S_{atl}^* \times S_{atl}$  is a function which defines the signature of the operators. The  $\diamond$  ('future'),  $\circ$  ('next'),  $\square$  ('always'), and  $U$  ('until') are temporal operators. The ATL model checker can be defined as the  $\Sigma_{atl}$ -language given in the form  $L_{atl} = \langle Sem_{atl}, Syn_{atl}, L_{atl}: Sem_{atl} \rightarrow Syn_{atl} \rangle$  where  $Syn_{atl}$  is the word algebra of the operator scheme  $\Sigma_{atl}$  generated by the operations from  $O_{atl}$  and a finite set of variables, representing atomic propositions, denoted by  $AP$ .  $Sem_{atl}$  represents ATL semantic algebra defined over the set of ATL formulas which are satisfied by the ATL model  $M$ .  $L_{atl}$  is a mapping which associates the set of satisfied formulas from  $Sem_{atl}$  to ATL expressions from  $Syn_{atl}$  which satisfy these formulas.

Having well-defined the ATL language, implementation of a ATL model checker will be equated with an algebraic compiler which translates a formula  $f$  of the ATL model to set of nodes  $Q'$  over which formula  $f$  is satisfied.

The Kripke structure is a natural "common-denominator" model for **closed systems**, independent of the high-level description of a system (given by example as a product of state machines).

In analogy, the natural "common-denominator" model for compositions of **open systems** is the *concurrent game structure*.

Thus, unlike CTL which is interpreted over Kripke structures, the ATL is interpreted over game structures. In order to capture compositions of *open systems*, we consider multi players games in which the set of players represents different component of the system and the environment.

The remainder of this paper is organized as follows. In section 2 we present a formal definition of concurrent game structures. In section 3 is presented the ATL logic with its syntax and semantics. A Java implementation of ATL model checker based on ANTLR is described in section 4. Conclusions are presented in section 5.

## 2 Concurrent game structures

Concurrent game structures can be used to model compositions of open systems. Unlike in Kripke structures, in a concurrent game structure, the environment is involved a state transition. The environment is modelled by a set of agents. Each agent may perform some actions and at least one action is available to the agent at each state.

### 2.1 Definition

A *concurrent game structure* is a tuple  $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$  with the following components [5]:

- A natural number  $k \geq 1$  of *players*. We identify the players with numbers  $1, \dots, k$ .
- A finite set  $Q$  of *states*
- A finite set  $\Pi$  of *propositions* (also called *observables*)
- For each state  $q \in Q$ , a set  $\pi(q) \subseteq \Pi$  of propositions true at  $q$ . The function  $\pi$  is called *labeling* (or *observation*) *function*.
- For each player  $a \in \{1, \dots, k\}$  and each state  $q \in Q$ , we identify the moves of player  $a$  at state  $q$  with the numbers  $1, \dots, d_a(q)$ , where  $d_a(q) \geq 1$  represents the number of available moves. For each state  $q \in Q$ , a *move vector* at  $q$  is a tuple  $\langle j_1, \dots, j_k \rangle$  such that  $1 \leq j_a \leq d_a(q)$  for each player  $a$ . Given a state  $q \in Q$ , we write  $D(q)$  for the set  $\{1, \dots, d_1(q)\} \times \dots \times \{1, \dots, d_k(q)\}$  of moves vector. The function  $D$  is called *move function*.
- For each state  $q \in Q$  and each move vector  $\langle j_1, \dots, j_k \rangle \in D(q)$ ,  $\delta(q, j_1, \dots, j_k) \in Q$  represents the state that results from state  $q$  if every player  $a \in \{1, \dots, k\}$  choose move  $j_a$ . The function  $\delta$  is called *transition function*.

The number of states of the structure  $S$  is  $n = |Q|$ . The *number of transitions* of  $S$  is  $m = \sum_{q \in Q} d_1(q) \times \dots \times d_k(q)$ , that is, the total number of elements in the move function  $D$ . Note that unlike in Kripke structures, the number of transitions is not bounded by  $n^2$ . For a fixed alphabet  $\Pi$  of propositions, the size of  $S$  is  $O(m)$ . [5]

We refer to a computation starting at state  $q$  as a *q-computation*. For a computation  $\lambda$  and a position  $i \geq 0$ , we use  $\lambda [i]$ ,  $\lambda [0, i]$ , and  $\lambda [i, \infty]$  to denote the  $i$ -th state of  $\lambda$ , the finite prefix  $q_0, q_1, \dots, q_i$  of  $\lambda$ , and the infinite suffix  $q_i, q_{i+1} \dots$  of  $\lambda$ , respectively. [5]

### 2.2 Example of a concurrent game structure

Consider a system with two processes,  $P_x$  and  $P_y$ . The process  $P_x$  assigns values to the Boolean variable  $x$ . When  $x = \text{false}$ , then  $P_x$  can leave the value of  $x$  unchanged or change it in *true*. When  $x = \text{true}$ , then  $P_x$  leaves the value of  $x$  unchanged. The process  $P_y$  assigns values to the Boolean variable  $y$ , in the same way as the process  $P_x$ .

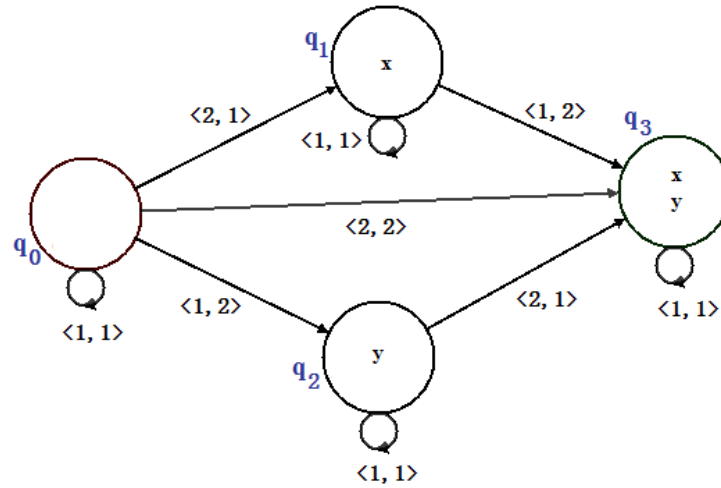


Fig. 1: An example of a concurrent game structure ( $S_{xy}$ )

We model the synchronous composition of two processes by the  $S_{xy}$  concurrent game structure, where  $S_{xy} = \langle k, Q, \Pi, \pi, d, \delta \rangle$

- $k=2$  (player 1 represents process  $P_x$ , player 2 represents process  $P_y$ )
- $Q = \{q_0, q_1, q_2, q_3\}$  –  $q_0$  means  $x=y=false$ ,  
 $q_1$  means  $x=true$  and  $y=false$ , etc.
- $\Pi = \{x, y\}$
- $\pi(q_0) = \emptyset$ ,  $\pi(q_1) = \{x\}$ ,  $\pi(q_2) = \{y\}$ ,  $\pi(q_3) = \{x, y\}$
- $d_1(q_0) = d_1(q_2) = 2$  (means in state  $q_0$  and  $q_2$  move 1 of player 1 leave the value of  $x$  unchanged, and move two changes the value of  $x$ )  
 $d_1(q_1) = d_1(q_3) = 1$  (means in state  $q_1$  and  $q_3$  player 1 has only one move, namely, to leave the value of  $x$  unchanged)  
 $d_2(q_0) = d_2(q_1) = 2$ ,  $d_2(q_2) = d_2(q_3) = 1$
- state  $q_0$  has four successors:  $\delta(q_0, 1, 1) = q_0$ ,  $\delta(q_0, 1, 2) = q_2$ ,  $\delta(q_0, 2, 1) = q_1$ ,  $\delta(q_0, 2, 2) = q_3$

A concurrent game is played on a state space. Every player chooses a move. The combination of choices determines a transition from the current state to a successor state.

The game structure presented in figure 1 is classified as a Moore synchronous game structure. That is, the state is partitioned according to the players. In each step, every player updates its own component of the state independently of the other players. The Moore subclass of concurrent game structures captures various notions of synchronous interaction between open systems.

### 3 ATL Logic

Alternating-time Temporal Logic (ATL) is a branching-time temporal logic that naturally describes computations of multi-agent system and multiplayer games. It offers selective quantification over program-paths that are possible outcomes of games [5].

#### 3.1. ATL syntax

The temporal logic ATL is defined with respect to a finite set  $\Pi$  of propositions and a finite set  $\Sigma = \{1, \dots, k\}$  of players.

An ATL formula is one of the following:

(S1)  $p$ , for propositions  $p \in \Pi$

(S2)  $\neg \varphi$  or  $\varphi_1 \vee \varphi_2$ , where  $\varphi$ ,  $\varphi_1$  and  $\varphi_2$  are ATL formulas

(S3)  $\langle\langle A \rangle\rangle \circ \varphi$ ,  $\langle\langle A \rangle\rangle \square \varphi$ , or  $\langle\langle A \rangle\rangle \varphi_1 \cup \varphi_2$ , where  $A \subseteq \Sigma$  is a set of players, and  $\varphi$ ,  $\varphi_1$  and  $\varphi_2$  are ATL formulas.

The operator  $\langle\langle \cdot \rangle\rangle$  is a path quantifier, and  $\circ$  ('next'),  $\square$  ('always'), and  $\cup$  ('until') are temporal operators. The logic ATL is similar to the branching time temporal logic CTL, only that path quantifiers are parameterized by sets of players. Sometimes we write  $\langle\langle a_1, \dots, a_k \rangle\rangle$  instead of  $\langle\langle \{a_1, \dots, a_k\} \rangle\rangle$ , and  $\langle\langle \cdot \rangle\rangle$  instead of  $\langle\langle \emptyset \rangle\rangle$ . Additional Boolean connectives are defined from  $\neg$  and  $\vee$  in the usual manner. Similar to CTL, we write  $\langle\langle A \rangle\rangle \diamond \varphi$  for  $\langle\langle A \rangle\rangle \text{true} \cup \varphi$ .

### 3.2. ATL semantics

Consider a game structure  $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ .

$\Sigma = \{1, \dots, k\}$  denote the set of players.

A *strategy* for player  $a \in \Sigma$  is a function  $f_a$  that maps every nonempty finite state sequence  $\lambda \in Q^+$  to a natural number such that if the last state of  $\lambda$  is  $q$ , then  $f_a(\lambda) \leq d_a(q)$ . Thus, the strategy  $f_a$  determines for every finite prefix  $\lambda$  of a computation a move  $f_a(\lambda)$  for player  $a$ . Each strategy  $f_a$  for player  $a$  induces a set of computations that player  $a$  can enforce.

Given a state  $q \in Q$ , a set  $A \subseteq \{1, \dots, k\}$  of players, and a set  $F_A = \{f_a \mid a \in A\}$  of strategies, one for each player in  $A$ , we define the *outcomes of  $F_A$  from  $q$*  to be the set  $out(q, F_A)$  of  $q$ -computations that the players in  $A$  enforce when they follow the strategies in  $F_A$ ;

A computation  $\lambda = q_0, q_1, q_2, \dots$  is in  $out(q, F_A)$  if  $q_0 = q$  and for all positions  $i \geq 0$ , there is a move vector  $\langle j_1, \dots, j_k \rangle \in D(q_i)$  such that

- $j_a = f_a(\lambda[0, i])$  for all players  $a \in A$ , and
- $\delta(q_i, j_1, \dots, j_k) = q_{i+1}$

Formal definition of ATL semantics is to consider a game structure  $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ . We write  $S, q \models \varphi$  to indicate that the state  $q$  satisfies the formula  $\varphi$  in the structure  $S$ . When  $S$  is clear from the context, we omit it and write  $q \models \varphi$ .

The satisfaction relation  $\models$  is defined, for all states  $q$  of  $S$  inductively as follows:

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

ATL can naturally express properties of open system [5]. Properties is the absence of deadlocks, where deadlock state is one in which a thread, say  $t$ , is permanently blocked from accessing a critical section.

In the following is described this requirement using the CTL formula, respectively ATL formula.

The CTL formula only asserts that it is always possible for all threads to cooperate so that  $t$  can eventually read and write ("collaborative possibility")

$$\forall \square (\exists \diamond \text{read} \wedge \exists \diamond \text{write})$$

The ATL formula guarantees execution of the critical section by the thread  $t$ , *no matter what the other threads in the system do* ("adversarial possibility")

$$\forall \square (\langle\langle t \rangle\rangle \diamond \text{read} \wedge \langle\langle t \rangle\rangle \diamond \text{write})$$

The path quantifiers A, E of CTL can be expressed in ATL with  $\langle\langle\emptyset\rangle\rangle$ ,  $\langle\langle\Sigma\rangle\rangle$  respectively. As a consequence, the CTL duality axioms can be rewritten in ATL, and become validities in the basic semantics:  $\neg\langle\langle\Sigma\rangle\rangle\Diamond\varphi \leftrightarrow \langle\langle\emptyset\rangle\rangle\Box\neg\varphi$ ,  $\neg\langle\langle\emptyset\rangle\rangle\Diamond\varphi \leftrightarrow \langle\langle\Sigma\rangle\rangle\Box\neg\varphi$ , where the  $\Sigma \in \{1, \dots, k\}$  describe the set of agents.

## 4 An ANTLR – Java implementation of ATL model checker

### 4.1 ATL Symbolic Model Checking

Model checking is a technology often used for the automated system verification.

- The model checking algorithms are currently used as verification techniques implemented in varied programming environments.
- The verified system can be a physical system or a real-time concurrent program.
- A model checking tool can be used to verify if a given system satisfies a temporal logic formula.
- The model checking problem for ATL: given a game structure  $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$  and an ATL formula  $\varphi$  the task is to find the set of states in  $Q$  that satisfy  $\varphi$ .

In order to solve the ATL model checking problem we designed and implemented an algebraic compiler denoted with  $\mathcal{C}$ .

### 4.2 ATL algorithm

The algebraic compiler  $\mathcal{C}$  translates formula  $\varphi$  of the ATL model to set of nodes  $Q'$  over which formula  $\varphi$  is satisfied. That is,  $\mathcal{C}(\varphi) = Q'$  where  $Q' = \{q \in Q \mid q \models \varphi\}$ .

The implementation of the algebraic compiler  $\mathcal{C}$  is made in two steps.

- First, we need a syntactic parser to verify the syntactic correctness of a formula  $\varphi$ .
- Then, we should deal with the semantics of the ATL language, respectively with the implementation of the ATL operators from the set  $\{\neg, \vee, \wedge, \rightarrow, \Diamond, \Box, \circ, \square, U\}$ .

For implementation of the algebraic compiler we choose the ANTLR (*Another Tool for Language Recognition*). ANTLR is a compiler generator which takes as input a grammar - an exact description of the source language, and generates a recognizer for the language defined by the grammar.

The algebraic compiler  $\mathcal{C}$  implements the following ATL symbolic model checking algorithm:

```

Function EvalA( $\varphi$ ) as set of states  $\subseteq Q$ 
  case  $\varphi = p$ :
    return [p];
  case  $\varphi = \neg\theta$ :
    return  $Q \setminus \text{Eval}_A(\theta)$ ;
  case  $\varphi = \theta_1 \vee \theta_2$ :
    return  $\text{Eval}_A(\theta_1) \cup \text{Eval}_A(\theta_2)$ ;
  case  $\varphi = \theta_1 \wedge \theta_2$ :
    return  $\text{Eval}_A(\theta_1) \cap \text{Eval}_A(\theta_2)$ ;
  case  $\varphi = \theta_1 \rightarrow \theta_2$ :
    return  $(Q \setminus \text{Eval}_A(\theta_1)) \cup \text{Eval}_A(\theta_2)$ ;
  case  $\varphi = \langle\langle A \rangle\rangle \circ \theta$ :
    return Pre(A, EvalA( $\theta$ ));
  case  $\varphi = \langle\langle A \rangle\rangle \Box \theta$ :
     $\rho := Q$ ;  $\tau := \text{Eval}_A(\theta)$ ;  $\tau_0 := \tau$ ;
    while  $\rho \not\subseteq \tau$  do
       $\rho := \tau$ ;
       $\tau := \text{Pre}(A, \rho) \cap \tau_0$ ;

```

```

        wend
        return  $\rho$ ;
    case  $\varphi = \langle\langle A \rangle\rangle \theta_1 \cup \theta_2$ :
         $\rho := \emptyset$ ;  $\tau := \text{Eval}_A(\theta_2)$ ;  $\tau_0 := \text{Eval}_A(\theta_1)$ ;
        while  $\rho \not\subseteq \tau$  do
             $\rho := \rho \cup \tau$ ;
             $\tau := \text{Pre}(A, \rho) \cap \tau_0$ ;
        wend
    return  $\rho$ ;
End Function

```

The  $\text{Pre}(A, \rho)$  function, where  $A \subseteq \Sigma$  and  $\rho \subseteq Q$ , returns the set of states  $q$  such that from  $q$ , the players in  $A$  can cooperate and enforce the next state to be in  $\rho$ .

$\text{Pre}(A, \rho)$  contains state  $q \in Q$  if for every player  $a \in A$  there exists a move  $j_a \in \{1, \dots, d_a(q)\}$  such that for all players  $b \in \Sigma \setminus A$  whatever are their moves we have  $\delta(q, j_1, \dots, j_k) \in \rho$

In order to translate a formula  $\varphi$  of an ATL model to the set of nodes  $Q'$  over which formula  $\varphi$  is satisfied, is necessary the attachment of specific actions to grammatical constructions within specification grammar of ATL.

The actions are written in target language of the generated parser (in our case, Java). These actions are incorporated in source code of the parser and are activated whenever the parser recognizes a valid syntactic construction in the translated ATL formula. In case of the algebraic compiler  $\mathcal{C}$ , the actions define the semantics of the ATL model checker, i.e., the implementation of the *ATL* operators.

The model checker generated by ANTLR from our specification grammar of ATL, takes as input the concurrent game structure  $S$  and formula  $\varphi$ , and provides as output the set  $Q' = \{q \in Q \mid q \models \varphi\}$  – the set of states where the formula  $\varphi$  is satisfied.

The corresponding *action* included in the ANTLR grammar of *ATL* language for implementing the  $\square$  operator is:

```

'<<A>> #' f=formula
{
    HashSet r = new HashSet(all_SetS);
    HashSet p = $f.set;
    while (!p.containsAll(r))
    {
        r = new HashSet(p);
        p = Pre(r);
        p.retainAll($f.set);
    }
    $set = r;
    trace("atlFormula", 4);
    printSet("<<A>>#" + $f.text, r);
}

```

Fig. 2: ANTLR implementation of  $\square$  operator

For ATL operator  $\square$ , in ANTLR we use the  $\#$  symbol.

In our implementation the `all_Set` is  $Q$ , and means all the state from model. The *formula* represents a term from a production of the ATL grammar and  $p, r, f$  variables are sets used in internal implementation of the algebraic compiler.

The  $\text{Pre}(r)$  is a function that returns the set of states  $p$  such that from  $p$ , the players in  $A$  can cooperate and enforce the next state to be in  $r$ .

The code from figure 2 represents the implementation of the  $\square$  ATL operator which is described in the symbolic model checking algorithm as:

```

Function EvalA(φ) as set of states ⊆ Q
...
case φ = <<A>>□θ:
    ρ:=Q; τ:= EvalA(θ); τ0:= τ;
    while ρ ⊄ τ do
        ρ := τ;
        τ:=Pre(A, ρ)∩τ0;
    wend
    return ρ;
...
endfunction
    
```

Fig. 3 The □ ATL operator from the model checking algorithm

In figure 4 is represented the algebraic compiler implementation process, based on our specification grammar of ATL language.

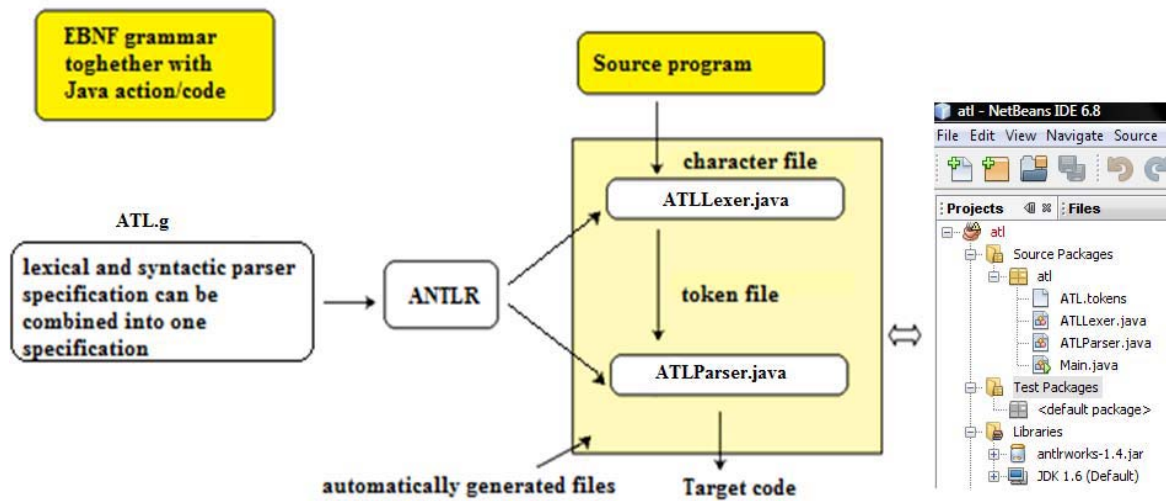


Fig. 4 Algebraic compiler implementation

For verification of formula  $\phi = \langle\langle A \rangle\rangle \square (x \vee y)$  we can use the ANTLR debugging facility to visualize the Abstract Syntactic Tree (AST), presented in the figure 5.

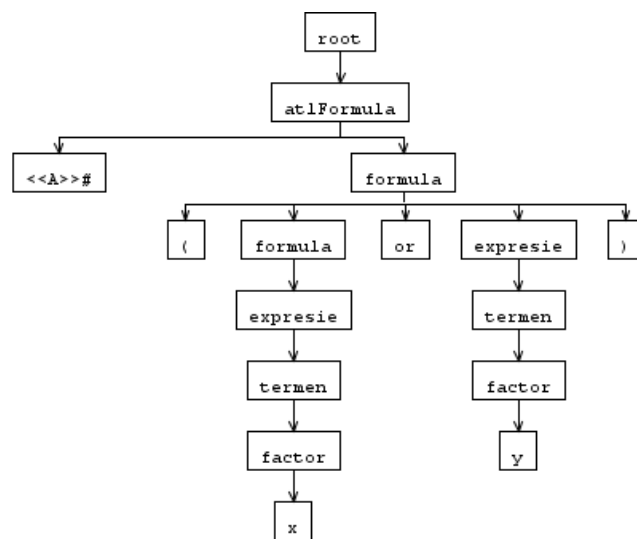
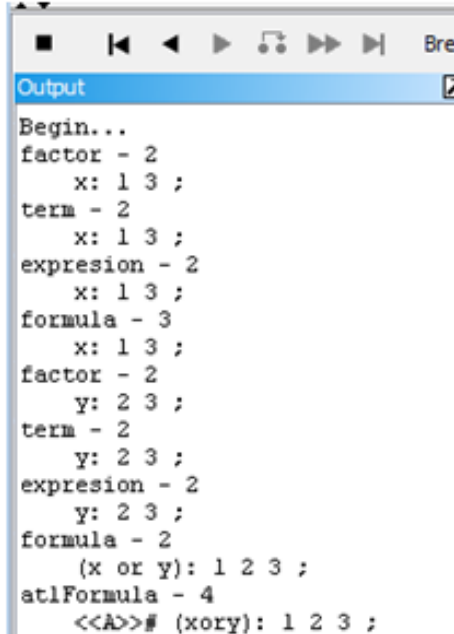


Fig. 5: Abstract Syntactic Tree (AST)



The AST is decorated with actions automatically executed when the parser recognizes syntactic components of formula  $\varphi$ . These actions implement the algebraic compiler  $\mathcal{C}$ . The output of  $\mathcal{C}$  is presented in figure 6.

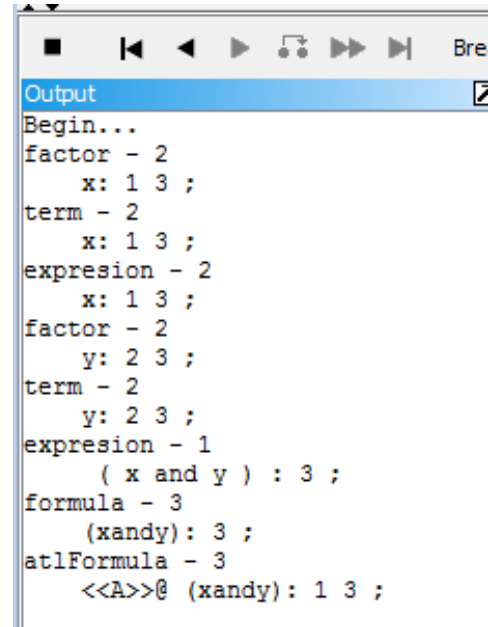
Given the ATL formula  $\varphi = \langle\langle A \rangle\rangle \circ (x \wedge y)$  for game structure from figure 1 with  $A = \{2\}$ , the output of the model checker is  $Q' = \{1, 3\}$ . From state  $q_1$  if player 2 chooses the move 2 the next state is  $q_3$  whatever is the move selected by the player 1. From the state  $q_2$  for the move 1 of the player 2, the player 1 can choose the move 1. Thus the game remains in state  $q_2$ . For that reason the state  $2 \notin Q'$ .



```

Output
Begin...
factor - 2
  x: 1 3 ;
term - 2
  x: 1 3 ;
expression - 2
  x: 1 3 ;
formula - 3
  x: 1 3 ;
factor - 2
  y: 2 3 ;
term - 2
  y: 2 3 ;
expression - 2
  y: 2 3 ;
formula - 2
  (x or y): 1 2 3 ;
atlFormula - 4
  <<A>># (xory): 1 2 3 ;
    
```

Fig. 6: The output of compiler  $\mathcal{C}$  for ATL formula  $\langle\langle A \rangle\rangle \square (x \vee y)$



```

Output
Begin...
factor - 2
  x: 1 3 ;
term - 2
  x: 1 3 ;
expression - 2
  x: 1 3 ;
factor - 2
  y: 2 3 ;
term - 2
  y: 2 3 ;
expression - 1
  ( x and y ) : 3 ;
formula - 3
  (xandy): 3 ;
atlFormula - 3
  <<A>>@ (xandy): 1 3 ;
    
```

Fig. 7: Output of the model checker for ATL formula  $\langle\langle A \rangle\rangle \circ (x \wedge y)$

## 5 Conclusion

In this article we built a CTL model checking tool, based on robust technologies (Java, ANTLR)

As a great facility we mention the capability of interactive debugging / visualization of the execution of the symbolic model checking algorithm.

The ATL algebraic compiler based on Java code generated by ANTLR using an original ATL grammar provides error-handling for eventual lexical/syntax errors in formula to be translated.

## References

- [1] Laura F. Cacovean, Florin Stoica, Dana Simian, *A New Model Checking Tool*, Proceedings of the European Computing Conference (ECC '11), Paris, France, April 28-30, 2011, pp. 358-364, ISBN: 978-960-474-297-4, ISSN: 2222-7342
- [2] L. Cacovean, F. Stoica, *Algebraic Specification Implementation for CTL Model Checker Using ANTLR Tools*, 2008 WSEAS International Conferences, Computers and Simulation in Modern Science - Volume II, Bucharest, Romania, 2008, pp. 45-50

- [3] M. Huth, M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, 2000.
- [4] L. F. Căcovăan, *Using CTL Model Checker for Verification of Domain Application Systems*, Proceedings of the 11th WSEAS International Conference on EC, Iași, Romania, 2010, pp. 262-267.
- [5] R. Alur, T. A. Henzinger, O. Kupferman, *Alternating-Time Temporal Logic*, Journal of the ACM, Vol. 49, No. 5, September 2002, pp. 672–713

LAURA FLORENTINA STOICA  
Department of Computer Science  
“Lucian Blaga” University of Sibiu  
Str. Dr. Ion Ratiu 5-7, 550012, Sibiu  
ROMANIA  
E-mail: laura.cacovean@ulbsibiu.ro

FLORIN STOICA  
Department of Computer Science  
“Lucian Blaga” University of Sibiu  
Str. Dr. Ion Ratiu 5-7, 550012, Sibiu  
ROMANIA  
E-mail: florin.stoica@ulbsibiu.ro