Implementing an ATL Model Checker tool using Relational Algebra concepts

Florin Stoica  
Department of Computer Science  
Faculty of Sciences, “Lucian Blaga” University of Sibiu  
Sibiu, Romania  
florin.stoica@ulbsibiu.ro

Laura Florentina Stoica  
Department of Computer Science  
Faculty of Sciences, “Lucian Blaga” University of Sibiu  
Sibiu, Romania  
laura.cacovean@ulbsibiu.ro

Abstract—Alternating-Time Temporal Logic (ATL) is a branching-time temporal logic that naturally describes computations of open systems. An open system interacts with its environment and its behavior depends on the state of the system as well as the behavior of the environment. ATL model-checking is a well-established technique for verifying that a formal model representing such a system satisfies a given property. In this paper we describe a new interactive model checker environment based on algebraic approach. Our tool is implemented in client-server paradigm. The client part allows an interactive construction of ATL models represented by concurrent game structures as directed multi-graphs. The server part embeds an ATL model checking algorithm implemented using ANTLR and Relational Algebra expressions translated into SQL queries. The server side of our tool was published as a Web service exposing its functionality to various clients through standard XML interfaces.

Keywords—ATL model checking; ANTLR; SQL; Web services; Relational Algebra

I. INTRODUCTION

Model checking is a technology widely used for the automated system verification and represents a technique for verifying that finite state systems satisfy specifications expressed in the language of temporal logics.

Main concern of formal methods in general, and model checking in particular, is helping to design correct systems. Detecting and eliminating bugs as early in the design cycle as possible is clearly an economic imperative.

The compositional modelling and design of reactive systems requires each component to be viewed as an open system [9].

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, the Alternating-time Temporal Logic (ATL) was defined [1]. ATL represents an extension of computation tree logic (CTL), which is interpreted over concurrent game structures (CGS).

ATL defines cooperation modalities of the form \( \langle \langle A \rangle \rangle \varphi \) (where \( A \) is a group of agents). Informally, \( \langle \langle A \rangle \rangle \varphi \) means that agents \( A \) have a collective strategy to enforce \( \varphi \), regardless of the actions of all the other agents [6].

The model checking problem for ATL is to determine whether a given model satisfies a given ATL formula.

A. Related work

ATL has been implemented in several symbolic tools for the analysis of open systems.

MOCHA [2] is a verification environment for the modular verification of heterogeneous systems. The input language of MOCHA is a machine readable variant of reactive modules. Reactive modules provide a semantic glue that allows the formal embedding and interaction of components with different characteristics [2].

MCMAS [10] is a symbolic model checker specifically tailored to agent-based specifications and scenarios. MCMAS supports specifications based on CTL and ATL, implements OBDD-based algorithms optimized for interpreted systems and supports fairness, counter-example generation, and interactive execution (both in explicit and symbolic mode). MCMAS has been used in a variety of scenarios including web-services, diagnosis, and security.

MCMAS takes a dedicated programming language called ISPL (Interpreted Systems Programming Language) as model input language. An ISPL file fully describes a multi-agent system (both the agents and the environment).

B. Comparing Symbolic and Explicit Model Checking

Two most common methods of performing model checking are explicit enumeration of states of the model and respectively the use of symbolic methods.

Symbolic model checkers analyse the state space symbolically using Ordered Binary Decision Diagrams (OBDDs), which were introduced in [11]. The binary decision diagram is a data structure for representing Boolean functions. With appropriate labelling of each state of the CGS structure, any expression on the Boolean variables represents a set of states of the structure. In contrast with explicit-state model checking, states in symbolic model checking are represented implicitly, as a solution to a logical equation. This approach saves space in memory since syntactically small equations can represent comparatively large sets of states [12]. A symbolic
model checker represents the CGS structure itself symbolically using OBDDs to represent transition relations by Boolean expressions. The key to symbolic model checking is to perform all calculations directly using these Boolean expressions, rather than using the CGS structure explicitly.

An efficient representation of the CGS structures using OBDDs can potentially allow much larger structures to be checked.

In the following we will present a short justification for the choice of the explicit-state model technique.

In [13] is presented a comparison between RULEBASE, a symbolic model checker developed at IBM Haifa Research Laboratory and the explicit LTL (Linear Temporal Logic) model checker SPIN [14]. The software verified was a Laboratory and the explicit LTL (Linear Temporal Logic) to manage 10^150 states. On the other hand, because of the limit it requires, in order to scaling its verification ability to handle real-world applications. Our tool is based on Web services technology to address the time constraints in verification of large models.

In this paper we will present a model-checking algorithm based on procedure from [1]. For a set \( A \) of agents and a set \( \Theta \) of states, implementation of almost all ATL operators imply the computation of function \( \text{Pre}(A, \Theta) \) – the set of states from which agents \( A \) can enforce the system into some state in \( \Theta \) in one move [2]. Our main contribution presented in this paper is the implementation of function \( \text{Pre}(A, \Theta) \) using Relational Algebra expressions, translated then into SQL statements. Other original approach is represented by the generation of an ATL model checker using ANTLR (Another Tool for Language Recognition) from our specification grammar of ATL.

The ATL semantics is implemented in our model checker tool by attaching of specific actions to grammatical constructions within specification grammar of ATL. The actions are written in target language of the generated parser, in this 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.

The paper is organized as follows. In section 2 we present the definition of the concurrent game structure. Section 3 defines the ATL syntax, and section 4 contains ATL semantics. In section 5 is described the implementation of an ATL model checker in ANTLR. In section 6 we introduce some relational algebra concepts which are used in our implementation of the ATL model checking algorithm. These concepts are applied in section 7. A performance analysis of our ATL model checker is made in section 8. Conclusions are presented in section 9.

II. THE CONCURRENT GAME STRUCTURE

A concurrent game structure is defined in [1] as a tuple \( S=(\Lambda, Q, \Gamma, \gamma, M, d, \delta) \) with the following components: a nonempty finite set of all agents \( \Lambda = \{1, \ldots, k\} \); a finite set of states \( Q \); a finite set of propositions (or observables) \( \Gamma \); the labelling (or observation) function \( \gamma \); a nonempty finite set of moves \( M \); the alternative moves function \( d \) and the transition function \( \delta \). For each state \( q \in Q \), \( \gamma(q) \subseteq \Gamma \) is the set of propositions true at \( q \). For each player \( a \in \{1, \ldots, k\} \) and each state \( q \in Q \), the alternative moves function \( d: \Lambda \times Q \rightarrow 2^M \) associates the set of available moves of agent \( a \) at state \( q \). In the following, the set \( d(a,q) \) will be denoted by \( d_a(q) \). For each state \( q \in Q \), a tuple \( j_1, \ldots, j_k \) such that \( j_a \in d_a(q) \) for each player \( a \in A \), represents a move vector at \( q \). We define the move function \( D: Q \rightarrow 2^M \), with \( M \) the set of all move vectors such that \( D(q) \subseteq d_1(q) \times \cdots \times d_k(q) \) is the set of move vectors at \( q \). We write

\[ D_q = \bigcup_{q \in Q} d_a(q) \tag{1} \]

for the set of available moves of agent \( a \) within the game structure \( S \).

The transition function \( \delta(q, j_1, \ldots, j_k) \), associates to each state \( q \in Q \) and each move vector \( (j_1, \ldots, j_k) \in D(q) \) the state that results from state \( q \) if every player \( a \in \{1, \ldots, k\} \) chooses move \( j_a \).
A computation of \( S \) is an infinite sequence \( \lambda = q_0, q_1, \ldots \) such that \( q_{i+1} \) is the successor of \( q_i \), \( \forall i \geq 0 \) [1]. A q-computation is a computation starting at state \( q \).

For a computation \( \lambda \) and a position \( i \geq 0 \), we denote by \( \lambda [i] \), \( \lambda [0,i] \), and \( \lambda [i,\infty] \) the \( i \)-th state of \( \lambda \), the finite prefix \( q_0, q_1, \ldots, q_i \) of \( \lambda \), and the infinite suffix \( q_i, q_{i+1}, \ldots \) of \( \lambda \), respectively [1].

### III. ATL SYNTAX

We denote by \( \mathcal{F}(\mathcal{A}) \) the set of all syntactically correct ATL formulas, defined over a concurrent game structure \( S \) and a set of agents \( \mathcal{A} \subseteq \Lambda \).

Each formula from \( \mathcal{F}(\mathcal{A}) \) can be obtained using the following rules:

1. (R1) if \( p \in \Gamma \) then \( p \in \mathcal{F}(\mathcal{A}) \);
2. (R2) if \( \{ \phi, \phi_1, \phi_2 \} \subseteq \mathcal{F}(\mathcal{A}) \) then \( \{ \top, \phi_1 \lor \phi_2 \} \subseteq \mathcal{F}(\mathcal{A}) \);
3. (R3) if \( \{ \phi, \phi_1, \phi_2 \} \subseteq \mathcal{F}(\mathcal{A}) \) then \( \{ (\langle \alpha \rangle) \circ \phi, (\langle \alpha \rangle) \land \phi \} \subseteq \mathcal{F}(\mathcal{A}) \), where \( \langle \alpha \rangle \)

The logic ATL is similar to the branching time temporal logic CTL, with the path quantifiers parameterized by sets of players from \( \Lambda \). The operator \( \langle \rangle \) is a path quantifier, and \( \circ \) ("next"), \( \diamond \) ("always"), \( \lozenge \) ("future") and \( U \) ("until") are temporal operators. A formula \( \langle \alpha \rangle \phi \) expresses that the team \( \mathcal{A} \) has a collective strategy to enforce \( \phi \) [5]. Boolean connectives can be defined from \( \top \) and \( \bot \) in the usual way.

ATL formula \( \langle \alpha \rangle \phi \) is true in a state if there exists a strategy \( \mathcal{A} \) such that the players in \( \mathcal{A} \) enforce \( \alpha \) when following the strategies from \( \mathcal{A} \).

### IV. ATL SEMANTICS

Consider a game structure \( S = (\Lambda, Q, \Gamma, \gamma, M, d, \delta) \) with \( \Lambda = \{ 1, \ldots, k \} \) the set of players.

A strategy for player \( a \in \Lambda \) is a function \( f_a : Q^+ \rightarrow D_a \) that maps every nonempty finite state sequence \( \lambda = q_0 q_1 q_2 \ldots \), to a move of agent \( a \) denoted by \( f_a(\lambda a) \in D_a \subseteq M \). Thus, the strategy \( f_a \) determines for every finite prefix \( \lambda \) of a computation a move \( f_a(\lambda) \) for player \( a \) in the last state of \( \lambda \).

Given a set \( \mathcal{A} \subseteq \{ 1, \ldots, k \} \), the set of all strategies of agents from \( \mathcal{A} \) is denoted by \( F_{\mathcal{A}} = \{ f_a \mid a \in \mathcal{A} \} \).

The outcome of \( F_{\mathcal{A}} \) is defined as \( \text{out}_{F_{\mathcal{A}}} : Q \rightarrow \Sigma(Q) \), where the function \( \text{out}_{F_{\mathcal{A}}} \) represents q-computations that the players from \( \mathcal{A} \) are enforcing when they follow the strategies from \( F_{\mathcal{A}} \).

A computation \( \lambda = q_0 q_1 q_2 \ldots \) is in \( \text{out}(q, F_{\mathcal{A}}) \) if \( q_0 = q \) and for all positions \( i \geq 0 \), there is a move vector \( j_0 j_1 \ldots j_i \in D(q) \) such that [1):

- \( f_a = f_a(\lambda [0,i]) \) for all players \( a \in \mathcal{A} \), and
- \( \delta(q_i j_0, \ldots, j_{i-1}) = q_{i+1} \).

For a game structure \( S \), we write \( \phi \models \varphi \) to indicate that the formula \( \varphi \) is satisfied in the state \( q \) of the structure \( S \).

For each state \( q \) of \( S \), the satisfaction relation \( \models \) is defined inductively as follows:

- for \( p \in \Gamma \), \( q \models p \Rightarrow p \in \gamma(q) \)
- \( q \models \neg \varphi \Leftrightarrow q \not\models \varphi \)
- \( q \models \varphi \lor \psi \Leftrightarrow q \models \varphi \) or \( q \models \psi \)
- \( q \models (\langle \alpha \rangle) \rightarrow \varphi \Leftrightarrow \) there exists a set \( F_{\mathcal{A}} \) of strategies, such that for all computations \( \lambda \in \text{out}(q, F_{\mathcal{A}}) \), we have \( \lambda[1] \models \varphi \) (the formula \( \varphi \) is satisfied in the successor of \( q \) within computation \( \lambda \)).
- \( q \models (\langle \alpha \rangle) \circ \varphi \Leftrightarrow \) there exists a set \( F_{\mathcal{A}} \) of strategies, such that for all computations \( \lambda \in \text{out}(q, F_{\mathcal{A}}) \), and all positions \( i \geq 0 \), we have \( \lambda[i] \models \varphi \) (the formula \( \varphi \) is satisfied in all states of computation \( \lambda \)).
- \( q \models (\langle \alpha \rangle) \varphi \Leftrightarrow \) there exists a set \( F_{\mathcal{A}} \) of strategies, such that for all computations \( \lambda \in \text{out}(q, F_{\mathcal{A}}) \), there exists a position \( i \geq 0 \) such that \( \lambda[i] \models \varphi \) and for all positions \( 0 \leq i < q \), we have \( \lambda[1] \models \varphi \).

### V. IMPLEMENTATION OF A MODEL CHECKER IN ANTLR

From a formal point of view, implementation of an ATL model checker will be accomplished through the implementation of an algebraic compiler [8] \( \mathcal{C} \) which translates the ATL formula \( \phi \) to the set of nodes \( Q^{+} \) over which formula \( \varphi \) is satisfied. That is, \( \mathcal{C}(\phi) = Q^{+} \) where \( Q^{+} = \{ q \in Q \mid q \models \varphi \} \).

We choose the ANTLR (Another Tool for Language Recognition) for implementation of the algebraic compiler. ANTLR [7] is a compiler generator which takes as input a grammar, and generates a recognizer for the language defined by the grammar.

Translation of a formula \( \varphi \) of an ATL model to the set of nodes \( Q^{+} \) over which formula \( \varphi \) is satisfied is accomplished by attaching of specific actions to grammatical constructions within specification grammar of ATL. These actions are written in Java, the target language of the generated parser. When ANTLR generates code using our ATL grammar as input, these actions are incorporated in the source code of the parser and are activated whenever the parser recognizes a valid syntactic construct from the translated ATL formula. In case of the algebraic compiler \( \mathcal{C} \), the attached 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 ATL specification grammar takes as input the model (a concurrent game structure) and the formula to be verified, and provides as output the set of states where the formula is satisfied.

Algorithm 1. ATL model checking algorithm

**Input:** the concurrent game structure \( S \) and the formula \( \varphi \)

**Output:** \( Q = \{ q \in Q \mid q \models \varphi \} \) – the set of states where the formula \( \varphi \) is satisfied.

**function** EvalA(\( \varphi \)) as set of states \( \subseteq Q \)
case \(\varphi \land \rho \): return \(\{ q \in Q \mid \rho \models \varphi(q) \} \);

[...]

VI. RELATIONAL ALGEBRA CONCEPTS

In the following we present all Relational Algebra concepts used in our algorithm described in the next section. More detailed aspects can be found in [3].

In order to introduce the following definitions, we assume that a set \(D\) of data types is given, and for each \(D \in D\), the set of possible values of data type \(D\) is denoted by \(val(D)\), which is also known as the domain of \(D\). A relation schema \(RS\) defines a (finite) sequence \(A_1, \ldots, A_n\) of distinct attribute names. The set of given attribute names will be denoted by \(\mathcal{A} = \{ A_1, \ldots, A_n \}\). Each attribute \(A_i\) has a data type \(D_i\), and a set of possible values represented by \(val(D_i)\), \(k = 1, n\). A relation schema \(RS\) may be written as \(RS = (A_1 : D_1, \ldots, A_n : D_n)\).

A tuple \(t\) with respect to the relation schema \(RS\) is \((A_1 : d_1, \ldots, A_n : d_n)\) a sequence \((d_1, \ldots, d_n)\) of \(n\) values such that \(d_i \in val(D_i)\), \(i = 1 \ldots n\). Relations are sets of tuples.

The Relational Algebra (RA) consists from the set of all finite relations over which are defined some operations. A query is an expression in the RA. The operations of RA can be nested to arbitrary depth such that complex queries can be evaluated. The final result will be a relation.

For the purpose of this paper, in the following we present from the set of RA operations only selection, projection (with renaming) and cartesian product.

The selection is denoted by \(\sigma\) and is parameterized by a simple predicate \(\alpha\). The operation \(\sigma_{\alpha}\) acts like a filter and selects a subset of the tuples of a relation, namely those which satisfy the predicate \(\alpha\).

For a given relation \(R\) and a predicate \(\alpha\) the expression \(\sigma_{\alpha}(R)\) corresponds to the following SQL query: \(\text{select distinct } * \text{ from } R \text{ where } \alpha\).

The projection \(\pi\) eliminates all attributes of the input relation excepting those mentioned in the list \(L\). If \(L = A_1, \ldots, A_n\), the projection \(\pi(L)\) produces for each input tuple \((A_1 : d_1, \ldots, A_n : d_n)\) an output tuple \((A_1 : d_1, \ldots, A_n : d_n)\).

There are two useful generalized projection operators. The first one is used to provide attribute renaming: \(\pi_{A_1 \mapsto B_1, \ldots, A_n \mapsto B_n}(R)\) provide for each input tuple \((A_1 : d_1, \ldots, A_n : d_n)\) an output tuple \((B_1 : d_1, \ldots, B_n : d_n)\).

The second generalized \(\pi\) operator is using computations to derive the values in new columns: \(\pi_{\text{EMAIL-Account \| @ \| Domain}(E)}\).

The relational algebra expression \(\pi_{A_1 \mapsto B_1, \ldots, A_n \mapsto B_n}(R)\) corresponds to the SQL query: \(\text{select distinct } A_1, A_2, \ldots, A_n \text{ from } R\) and for the expression \(\pi_{A_1 \mapsto B_1, \ldots, A_n \mapsto B_n}(R)\) the equivalent SQL query is \(\text{select distinct } A_1, A_2, \ldots, A_n \text{ from } R\).

In general, queries need to combine values from several relations. In RA, such queries are formulated using the Cartesian product, denoted by symbol \(\times\). The Cartesian

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

```java
'<<A>> 
```
product \( R \times V \) of two relations \( R, V \) is computed by concatenating each tuple \( r \in R \) with each tuple \( v \in V \). If \( r = (A_1 : a_1, \ldots, A_n : a_n) \) and \( v = (B_1 : b_1, \ldots, B_m : b_m) \) then \( r \cdot v = (A_1 : a_1, \ldots, A_n : a_n, B_1 : b_1, \ldots, B_m : b_m) \) where \( \cdot \) denotes tuple concatenation. The attribute names must be unique within a tuple \( r \cdot v \). \( R \times V \) can be computed by the equivalent SQL query: \( \text{select } R, v \text{ from } R, V \).

The unique column name restriction is solved in SQL easily: a common attribute \( A \) of relations \( R \) and \( V \) may uniquely be identified by \( R.A \) respectively \( V.A \).

In \( RA \), this solution is formalized by the renaming operator \( \rho_A(R) \). If \( R \) is a relation with schema \( \text{sch}(R) = (A_1 : D_1, \ldots, A_n : D_n) \), then using the renaming operator \( \rho_A(R) = \pi_{x_1 \cdots x_n}(\pi_{y_1 \cdots y_n}(R)) \) is obtained a relation with schema \( (x.A_1 : D_1, \ldots, x.A_n : D_n) \).

Because the combination of Cartesian product and selection in queries is frequently, a special operator join has been introduced, denoted by \( \bowtie \). For a set \( A \) of agents, \( R \bowtie V = \sigma_Q(R \times V) \), where the join predicate \( Q \) may refer to attributes of \( R \) and \( V \).

The join operator combines tuples from two relations and acts like a filter, removing tuples without join partner. In SQL language, the relational algebra expression \( R \bowtie V \) can be written as: \( \text{select } \ast \text{ from } R \text{ join } V \text{ on } \theta \).

The left outer join operator denoted by \( \bowletimes \), preserves all tuples in its left argument, even if a tuple does not fit with a partner in the join.

**VII. USING RELATIONAL ALGEBRA IN MODEL CHECKING**

For a concurrent game structure \( S \) presented in section 2, can be defined a directed multi-graph \( G_S = (X, U) \), where \( X = Q \), and \( (b, e) \in U \iff \exists j_1, \ldots, j_k \in Q \) such as \( e = (b, j_1) \in U \). \( Q \) is the set of states. The labelling function for the graph \( G_S \) is defined as follows:

\( L : U \rightarrow M \), \( \forall \; u \in U \Rightarrow \exists \; \delta \in \{j_1, \ldots, j_k \} \in D(b) \) such as \( \delta(b,j_1,\ldots,j_k) = e \).

We define the relation schema \( (B : Q_b, M_1 : D_1, \ldots, M_k : D_k, E : Q_e) \) where \( Q_b = \{b \in Q \mid \exists e \in Q \text{ such as } (b, e) \in U \}, Q_e = \{e \in Q \mid \exists b \in Q \text{ such as } (b, e) \in U \} \) and \( D_i, i \in \{1, \ldots, k \} \) was defined in (1), such as if \( R_S \) is a relation name with schema defined above, \( (B : b, M : j_1, \ldots, M_k : j_k, E : e) \in R_S \iff (j_1, \ldots, j_k) = L((b, e)) \).

For a set \( \mathcal{A} = \{a_1, \ldots, a_m\} \) of agents, \( \mathcal{A} \subseteq \Lambda, \mathcal{A} = \{i_1, \ldots, i_m\} \), we define \( R_\mathcal{A}(\mathcal{A}) = \pi_{x_1 \cdots x_n}(\pi_{y_1 \cdots y_n}(R_S)) \) where \( i \in \mathcal{A}, j \in \{1, \ldots, m\} \) and \( R_\mathcal{A}(\mathcal{A}) = \pi_{B, \mathcal{A} : L \cdot \mathcal{A} \cdot \mathcal{A} \cdot M \cdot \mathcal{A} \cdot E}(\pi_{B, \mathcal{A} : B \cdot \mathcal{A} \cdot M \cdot \mathcal{A} \cdot E}(R_S)) \) where the operator \( \circ \) can be defined as follows: \( i \circ j = i \| j \| j \).

For a set \( \Theta \subseteq \mathcal{Q}_b, \Theta = \{b \in \mathcal{Q}_b \mid \exists \; e \in \Theta \} \), we can define as follows: \( \mathcal{A} \cap \Theta \subseteq \mathcal{Q}_b, \Theta = \{b \in \mathcal{Q}_b \mid \exists \; e \in \Theta \} \), such as \( (b, j_1, \ldots, j_k, e) \in \mathcal{A} \cap \Theta \).

With other words, \( b \in \mathcal{A} \cap \Theta \iff \exists \; j_i \in d_b^*(b), i_j \in \mathcal{A} \cap \Theta, l = 1, m \) and \( \exists \; e \in \Theta \) such as \( (b, j_1, \ldots, j_k, e) \in \mathcal{A} \cap \Theta \).

The above algorithm can be implemented in SQL language as follows:

**Algorithm 3. Computing \( \text{Pre}(\mathcal{A}, \Theta) \) function using SQL statements**

\[
\text{select distinct } B \text{ from } (\text{select distinct } x.B, y.LABEL \text{ from } (\text{select distinct } B, \text{LABEL from model where } E \in \Theta ) \text{ x } \text{ left join } (\text{select distinct } B, \text{LABEL from model where } E \notin \Theta ) \text{ y } \text{ on } x.B = y.B \text{ and } x.LABEL = y.LABEL \text{ where } y.LABEL \text{ is null } ) \text{ z }) \]

**VIII. PERFORMANCE ANALYSIS OF THE ATL MODEL CHECKER**

In this section we evaluate the effectiveness of our approach in designing and implementing an ATL model checker and we report some experimental results.

We encoded the Tic-Tac-Toe (TTT) game in the formalism of concurrent game structures. The game is played by two opponents with a turn-based modality on a 3×3 board. The two players take turns to put pieces on the board. A single piece is put for each turn and a piece once put does not move. A player wins the game by first lining three of his or her pieces in a straight line, no matter horizontal, vertical or diagonal.
We consider the computer playing TTT game with a user (human) and the ATL model checking algorithm is used to achieve a winning strategy for the computer.

Due to the relatively large size of the structure representing the ATL model at the first moves, playing the TTT game represents a good opportunity to study the impact of the used technologies in the performance of our model checker.

For this purpose, were used two different database servers to implement the Web service, namely MySql 5.5 and respectively H2 1.3. The H2 database supports the in-memory mode (the data is not persisted), well suited for high performance operations.

The major impact on performance of the ATL model checker is represented by the efficiency of the function Pre(), which was presented in detail in section 7. Its implementation is based exclusively on the database server used.

In table I are presented the results showing the performance of our ATL model checker related to the database server used:

<table>
<thead>
<tr>
<th>Number of states</th>
<th>MySQL 5.5 (seconds)</th>
<th>H2 1.3 (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4791</td>
<td>≈0.86</td>
<td>≈1.33</td>
</tr>
<tr>
<td>4255</td>
<td>≈1.62</td>
<td>≈1.17</td>
</tr>
<tr>
<td>3732</td>
<td>≈1.41</td>
<td>≈0.99</td>
</tr>
<tr>
<td>3423</td>
<td>≈1.24</td>
<td>≈0.90</td>
</tr>
<tr>
<td>3683</td>
<td>≈1.21</td>
<td>≈0.85</td>
</tr>
<tr>
<td>2307</td>
<td>≈0.86</td>
<td>≈0.58</td>
</tr>
<tr>
<td>2236</td>
<td>≈0.75</td>
<td>≈0.56</td>
</tr>
</tbody>
</table>

In [18] the Tic-Tac-Toe was implemented in the Reactive Modules Language (RML). RML is the model description language of the ATL model checker MOCHA, which was developed by Alur et al. (Alur et al., 1998). Experimental results showed that the time necessary to find a winning strategy for a player, on a configuration with a Dual-Core 1.8Ghz CPU, was 1 minute and 6 seconds. Running on the same configuration, our ATL checker tool is able to find a winning strategy in about 4 seconds using MySql as a database server and 3 seconds when H2 Server was used.

IX. CONCLUSION

In this paper we built an ATL model checking tool, based on Java code generated by ANTLR using an original ATL attribute grammar which implements the ATL semantics using semantic actions. The main contribution of this paper consists in implementation of ATL operators using Relational Algebra expressions translated into SQL queries.

The C# implementation of the client part of our new tool (ATL Designer) allows an interactive graphical specification of the ATL model as a directed multi-graph.

The server component of our tool (ATL Checker) was published as a Web service, exposing its functionality through standard protocols.

As the experimental results showed, the in-memory databases represent a promising technology for our ATL model checker.

The ATL Designer, a ready to be deployed Web Service package for Tomcat 7.x, and the ATL API Client libraries for Java and C# can be downloaded from http://use-it.ro.

REFERENCES