Scalability in Model Checking through Relational Databases

Florin Stoica *florin.stoica@ulbsibiu.ro*
Department of Mathematics and Computer Science
"Lucian Blaga" University
Sibiu, 550012, Romania

Abstract

In this paper we present a new ATL model checking tool used for verification 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. The Alternating-Time Temporal Logic (ATL) logic is interpreted over concurrent game structures, considered as natural models for compositions of open systems. In contrast to previous approaches, our tool permits an interactive design of the ATL models as state-transition graphs, and is based on client/server architecture: ATL Designer, the client tool, allows an interactive construction of the concurrent game structures as a directed multi-graphs and the ATL Checker, the core of our tool, represents the server part and is published as Web service. The ATL Checker includes an algebraic compiler which was implemented using ANTLR (Another Tool for Language Recognition). Our model checker tool allows designers to automatically verify that systems satisfy specifications expressed by ATL formulas. The original implementation of the model checking algorithm is based on Relational Algebra expressions translated into SQL queries. Several database systems were used for evaluating the system performance in verification of large ATL models.

Keywords: ATL, Model Checking, Web Services, Relational Algebra, SQL.

1. 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.

A Computation Tree Logic (CTL) specification is interpreted over Kripke structures, which are graph-like structures, in which nodes represent states and arcs represent transitions between states.

The set of all paths through a Kripke structure is assumed to correspond to the set of all possible computations of a system. CTL logic is branching-time logic, meaning that its formulas are interpreted over all paths beginning in a given state of the Kripke structure.

A CTL formula encodes properties that can occur along a particular temporal path as well as to the set of all possible paths. A path in a CTL model is interpreted as sequences of successive states of computations. The CTL syntax includes several operators for describing temporal properties of systems: *A* (for all paths), *E* (there is a path), ○ (at the next moment), ◊ (in future), □ (always) and *U* (until).

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

The branching time temporal logic CTL has a limited value when applied to open systems [1], although it can be used successfully for domain oriented applications [2]. 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 [3]. ATL represents an extension of CTL, which is interpreted over concurrent game structures (CGS).

ATL replaces path quantifiers *A* and *E* by cooperation modalities of the form $\langle\langle \mathcal{A} \rangle\rangle$ φ (where $\mathcal{A}$ is a group of agents). Informally, $\langle\langle \mathcal{A} \rangle\rangle$ φ means that agents $\mathcal{A}$ have a collective strategy to enforce φ, regardless of the actions of all the other agents [4].

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

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 [5]. 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 [6]. 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.

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

In [2] is presented a verification environment called MOCHA 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 [7].

In [8] is described MCMAS, 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).

The aim of our research was to develop a reliable, easy to maintain, scalable model checker tool to improve applicability of ATL model checking in design of general-purpose computer software.

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

In [9] 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 [10]. The software verified was a distributed storage subsystem software application. The

state space size handled by SPIN was 10^8 in a 3-process model. Using symbolic model checking, RULEBASE keeps a compressed representation of the state space and thus was able to manage 10^{150} states. On the other hand, because of the limit on state size, RULEBASE could not represent a state large enough to include the information needed for more than 2-process configuration [9].

Most hardware designs are based on a clocked-approach and thus are synchronous. For these systems, the symbolic model checking approach is more appropriate [11].

On the other hand, for nondeterministic, high-level models of hardware protocols, it has previously been argued that explicit model checking is better than symbolic model checking [12]; this is because the communication mechanisms inherent in protocols tend to cause the BDDs in symbolic model checking to blow up [13].

In their basic form, symbolic approaches tend to perform poorly on asynchronous models where concurrent interleaving are the main source of explosion of the BDD representation, and explicit-state model-checkers have been the preferred approach for such models [13].

Concurrent software is asynchronous as the different components might be running on different processors or be interleaved by the scheduler of the operating system. Taking into account the above considerations, in our tool we are using an explicit-state model technique.

The most pressing challenge in model checking today is scalability [6]. A model-checking tool must be efficient, in terms of the size of the models it can reason about and the time and space it requires, in order to scaling its verification ability to handle real-world applications.

An orthogonal approach to increase the capacity of an explicit-state model checker tool is to exploit the memory and computational resources of multiple computers in a distributed computing environment [13]. Following this idea, 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 [3]. For a set $\mathcal{A}$ of agents and a set Θ of states, implementation of almost all ATL operators imply the computation of function $Pre(\mathcal{A}, \Theta)$ – the set of states from which agents $\mathcal{A}$ can enforce the system into some state in Θ in one move [7]. Our main contribution presented in this paper is the implementation of function $Pre(\mathcal{A},\Theta)$ 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. In section 3 is defined 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. In section 8 are presented several examples of computations based on results obtained in section 7. In section 9 is described the architecture of our ATL model checker tool: the server part, published as a Web service and a GUI client developed in C#. A performance analysis of our ATL model checker is made in section 10. Conclusions are presented in section 11.

2. THE CONCURRENT GAME STRUCTURE

A *concurrent game structure* is defined in [3] as a tuple $S=\langle \Lambda, Q, \Gamma, \gamma, M, d, \delta \rangle$ 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*) Γ; the ***labelling*** (or *observation*) function γ; a nonempty finite set of moves M; the ***alternative moves*** function d and the ***transition function*** δ**.** 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 $d: \Lambda \times Q \to 2^M$ 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 $\langle j_1,\ldots,j_k \rangle$ such that $j_a \in d_a(q)$ for each player $a \in \Lambda$, represents a *move vector* at q. We define the ***move function*** $D : Q \to 2^{\overline{M}}$, with $\overline{M}$ the set of all move vectors such that $D(q) \subseteq d_1(q) \times \ldots \times d_k(q)$ is the set of move vectors at q. We write

$$D_a = \bigcup_{q \in Q} d_a(q) \tag{1}$$

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 $\langle j_1,\ldots,j_k \rangle \in D(q)$ 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, $\forall i \geq 0$ [3]. A *q-computation* is a computation starting at state q.

For a computation λ and a position $i \geq 0$, we denote by $\lambda[i]$, $\lambda[0,i]$, and $\lambda[i,\infty]$ the i-th state of λ, the finite prefix $q_0, q_1,\ldots,q_i$ of λ, and the infinite suffix $q_i, q_{i+1} \ldots$ of λ, respectively [3].

3. ATL SYNTAX

We denote by $\mathcal{F}_S(\mathcal{A})$ 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 $\mathcal{F}_S(\mathcal{A})$ can be obtained using the following rules:

(*R1*) if $p \in \Gamma$ then $p \in \mathcal{F}_S(\mathcal{A})$;

(*R2*) if $\{\varphi, \varphi_1, \varphi_2\} \subseteq \mathcal{F}_S(\mathcal{A})$ then $\{\neg \varphi, \varphi_1 \vee \varphi_2\} \subseteq \mathcal{F}_S(\mathcal{A})$;

(*R3*) if $\{\varphi, \varphi_1, \varphi_2\} \subseteq \mathcal{F}_S(\mathcal{A})$ then $\{\langle\langle \mathcal{A} \rangle\rangle \circ \varphi, \langle\langle \mathcal{A} \rangle\rangle \Box \varphi, \langle\langle \mathcal{A} \rangle\rangle \varphi_1 \, U \, \varphi_2\} \subseteq \mathcal{F}_S(\mathcal{A})$.

The logic ATL is similar to the branching time temporal logic CTL, with difference that path quantifiers are parameterized by sets of players from Λ. The operator $\langle\langle \, \rangle\rangle$ is a path quantifier, and $\circ$ (next), $\Diamond$ (future), $\Box$ (always) and U (until) are temporal operators. A formula $\langle\langle \mathcal{A} \rangle\rangle \varphi$ expresses that the team $\mathcal{A}$ has a collective strategy to enforce φ [14]. Boolean connectives can be defined from $\neg$ and $\vee$ in the usual way. The ATL formula $\langle\langle \mathcal{A} \rangle\rangle \Diamond \varphi$ is equivalent with $\langle\langle \mathcal{A} \rangle\rangle$ *true* $U \varphi$.

4. ATL SEMANTICS

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

A *strategy* for player $a \in \Lambda$ is a function $f_a: Q^+ \to D_a$ that maps every nonempty finite state sequence $\lambda = q_0, q_1, \ldots q_n$, $n \geq 0$, to a move of agent a denoted by $f_a(\lambda) \in D_a \subseteq M$. Thus, the strategy f_a determines for every finite prefix λ of a computation a move $f_a(\lambda)$ for player a in the last state of λ.

Given a set $\mathcal{A} \subseteq \{1,\ldots,k\}$ of players, 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 $out_{F_{\mathcal{A}}} : Q \to \mathcal{P}(Q^+)$, where $out_{F_{\mathcal{A}}}(q)$ represents *q-computations* that the players from $\mathcal{A}$ are enforcing when they follow the strategies from $F_{\mathcal{A}}$. In the following, for $out_{F_{\mathcal{A}}}(q)$ we will use the notation $out(q, F_{\mathcal{A}})$. A computation $\lambda=q_0,q_1,q_2,\ldots$ is in $out(q, F_{\mathcal{A}})$ if $q_0=q$ and for all positions $i \geq 0$, there is a move vector $\langle j_1,\ldots,j_k \rangle \in D(q_i)$ such that [3]:

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

For a game structure S, we write $q \models \varphi$ to indicate that the formula φ 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 $p \in \Gamma$, $q \models p \Leftrightarrow p \in \gamma(q)$
- $q \models \neg\varphi \Leftrightarrow q \not\models \varphi$
- $q \models \varphi_1 \vee \varphi_2 \Leftrightarrow q \models \varphi_1$ or $q \models \varphi_2$
- $q \models \langle\langle \mathcal{A} \rangle\rangle \circ \varphi \Leftrightarrow$ there exists a set $F_{\mathcal{A}}$ of strategies, such that for all computations $\lambda \in out(q, F_{\mathcal{A}})$, we have $\lambda[1] \models \varphi$ (the formula φ is satisfied in the successor of q within computation λ).
- $q \models \langle\langle \mathcal{A} \rangle\rangle \Box \varphi \Leftrightarrow$ there exists a set $F_{\mathcal{A}}$ of strategies, such that for all computations $\lambda \in out(q, F_{\mathcal{A}})$, and all positions $i \geq 0$, we have $\lambda[i] \models \varphi$ (the formula φ is satisfied in all states of computation λ).
- $q \models \langle\langle \mathcal{A} \rangle\rangle \varphi_1 \, U \, \varphi_2 \Leftrightarrow$ there exists a set $F_{\mathcal{A}}$ of strategies, such that for all computations $\lambda \in out(q, F_{\mathcal{A}})$, there exists a position $i \geq 0$ such that $\lambda[i] \models \varphi_2$ and for all positions $0 \leq j < i$, we have $\lambda[j] \models \varphi_1$.

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

5. 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 $\mathcal{C}$ in two steps [15].

First, we need a syntactic parser to verify the syntactic correctness of a formula φ. Second, we should deal with the semantics of the ATL language, respectively with the implementation of the ATL operators: ¬, ∨, ∧, →, ◊, ○, □ and *U*.

The algebraic compiler $\mathcal{C}$ translates the ATL formula φ to the set of nodes Q' over which formula φ is satisfied. That is, $\mathcal{C}(\varphi)=Q'$ 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 [16] 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 φ of an ATL model to the set of nodes Q' over which formula φ 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 construction in 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 concurrent game structure S and the formula φ, and provides as output $Q'=\{q\in Q/\ q \vDash \varphi\}$ – the set of states where the formula φ is satisfied.

The algebraic compiler $\mathcal{C}$ implements the following ATL model checking algorithm [17], [3]:

Algorithm 1. ATL model checking algorithm

Input: the concurrent game structure S and the formula φ
Output: $Q'=\{q\in Q|\ q \vDash \varphi\}$ – the set of states where the formula φ is satisfied.
function EvalA(φ) as set of states $\subseteq Q$
  case $\varphi=p$:
    return $[p] = \{q \in Q \mid p \in \gamma(q)\}$;
  case $\varphi= \neg\theta$:
    return $Q \setminus$ EvalA(θ);
  case $\varphi=\theta1\vee\theta2$:
    return EvalA($\theta1$)$\cup$ EvalA($\theta2$);
  case $\varphi=\theta1\wedge\theta2$:
    return EvalA($\theta1$) $\cap$ EvalA($\theta2$));
  case $\varphi=\theta1\rightarrow\theta2$:
    return ($Q \setminus$ EvalA($\theta1$)) $\cup$ EvalA($\theta2$);
  case $\varphi =\langle\langle\mathcal{A}\rangle\rangle\circ\theta$:
    return Pre($\mathcal{A}$,EvalA(θ));
  case $\varphi =\langle\langle\mathcal{A}\rangle\rangle\Box\theta$:
    $\rho:=Q$; $\tau:=$ EvalA(θ); $\tau0:=\tau$;
    while $\rho \not\subseteq \tau$ do
      $\rho := \tau$; $\tau:=$Pre($\mathcal{A}$, ρ)$\cap\tau0$;
    wend
    return ρ;
  case $\varphi = \langle\langle\mathcal{A}\rangle\rangle\ \theta1\ U\ \theta2$:
    $\rho:=\varnothing$; $\tau:=$ EvalA($\theta2$); $\tau0:=$ EvalA($\theta1$);
    while $\tau \not\subseteq \rho$ do
      $\rho := \rho \cup \tau$;
      $\tau:=$Pre($\mathcal{A}$, ρ)$\cap\tau0$;
    wend
    return ρ;
end function

The corresponding *action* included in the ANTLR grammar of ATL language for implementing the $\Box$ 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);
    }
    $set=r;
    trace("atlFormula");
    printSet("<<A>>#"+$f.text,r);
}
```

For the □ ATL operator we use in ANTLR the # symbol. Also, we denote the ◊ ATL operator with ~ symbol and the ○ operator is replaced by @ symbol.

In our implementation the *all_Set* is *Q*, and means all states of the model. The *formula* represents a term from a production of the ATL grammar, and *p*, *r*, *f* variables are sets used in the internal implementation of the algebraic compiler. Functions *trace()* and *printSet()* are debugging functions.

In case of large ATL models, with many states and agents, it is very important for the model checker tool to have an efficient implementation for *Pre*($\mathcal{A}$, ρ) function – the set of states from which agents $\mathcal{A}$ can enforce the system into some state in ρ in one move – which appears in several ATL operators. In the following, we made an original formalization of the *Pre*() function using Relational Algebra (RA) concepts. Then, we will translate the obtained relational algebra expression into a SQL statement, which represents a concise implementation of the *Pre*() function, ready to be executed using a very efficient query optimizer on a database server.

6. 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 [18].

In order to introduce the following definitions, we assume that a set $\mathcal{D}$ of data types is given, and for each $D \in \mathcal{D}$, 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_k has a data type D_k, and a set of possible values represented by $val(D_k)$, $k = \overline{1,n}$.

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 $RS = (A_1 : D_1, \ldots, A_n : D_n)$ is a sequence $(d_1, \ldots, d_n)$ of *n* values such that $d_i \in val(D_i)$, $i = \overline{1,n}$.

Relations are sets of tuples.

A relational database schema consists of a finite set of relation names $\mathcal{R}$, and for every relation $R \in \mathcal{R}$ is also considered its relation schema *sch*(*R*).

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 σ and is parameterized by a simple predicate ω. The operation σ_ω acts like a filter and selects a subset of the tuples of a relation, namely those which satisfy the predicate ω. The predicate ω has the form:

expr operator expr,

where *expr* is an expression built from attributes, constants, and data type operations (+, -, *, /, etc) and operator can be:

- =, ≠

- <, <=, >, >=
- data type-dependent predicates (LIKE, IN or $\in$, etc)

For a given relation *R* and a predicate ω, the expression $\sigma_\omega(R)$ corresponds to the following SQL query:

*select distinct * from R where* ω

More complex selection predicates may be constructed using the boolean connectives ($\vee,\wedge,\neg$).

The projection π_L eliminates all attributes of the input relation excepting those mentioned in the list *L*. If $L = A_{i_1},\ldots,A_{i_m}$ the projection $\pi_L(R)$ produces for each input tuple $(A_1 : d_1,\ldots,A_n : d_n)$ an output tuple $(A_{i_1} : d_{i_1},\ldots,A_{i_m} : d_{i_m})$.

There are two useful generalized projection operators. The first one is used to provide attribute renaming:

$$\pi_{B_1 \leftarrow A_{i_1},\ldots,B_m \leftarrow A_{i_m}}$$

The projection $\pi_{B_1 \leftarrow A_{i_1},\ldots,B_m \leftarrow A_{i_m}}(R)$ provide for each input tuple $(A_1 : d_1,\ldots,A_n : d_n)$ an output tuple $(B_1 : d_{i_1},\ldots,B_m : d_{i_m})$.

The second generalized π operator is using computations to derive the values in new columns:

$$\pi_{NAME,\,EMAIL \leftarrow Account\,||\,'@'\,||\,Domain}(EMPLOYEE)$$

where || operator represents string concatenation.

The relational algebra expression $\pi_{A_1,\ldots,A_m}(R)$ corresponds to the SQL query:

select distinct $A_1,\ldots,A_m$ *from R*

and for the expression $\pi_{B_1 \leftarrow A_1,\ldots,B_m \leftarrow A_m}(R)$ the equivalent SQL query is

select distinct A_1 *as* $B_1,\ldots,A_m$ *as* B_m *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 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

$$r \bullet v = (A_1 : a_1,\ldots,A_n : a_n, B_1 : b_1,\ldots,B_m : b_m)$$

where • denotes tuple concatenation. The attribute names must be unique within a tuple $r \bullet v$.
$R \times V$ can be computed by the equivalent SQL query:

select R., V.* 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 $\rho_x(R)$. If *R* is a relation with schema $sch(R) = (A_1 : D_1,\ldots,A_n : D_n)$, then $\rho_x(R) = \pi_{x.A_1 \leftarrow A_1,\ldots,x.A_n \leftarrow A_n}(R)$ is 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 $\bowtie_\theta$:

$$R \bowtie_\theta V \equiv \sigma_\theta (R \times V)$$

where the join predicate θ may refer to attribute names of R and V.

The join operator combines tuples from two relations and acts like a filter, removing tuples without join partner:

x.A	x.B
a_1	b_1
a_2	b_2
a_3	b_3

$\bowtie_{x.B=y.B}$

y.B	y.C
b_3	c_3
b_4	c_4
b_5	c_5

=

A	B	C
a_3	b_3	c_3

In SQL language, the relational algebra expression $R \bowtie_\theta V$ can be written as:

*select * from R join V on* θ.

The left outer join operator denoted by $\overset{\frown}{\bowtie}_\theta$, preserves all tuples in its left argument, even if a tuple does not fit with a partner in the join:

x.A	x.B
a_1	b_1
a_2	b_2
a_3	b_3

$\overset{\frown}{\bowtie}_{x.B=y.B}$

y.B	y.C
b_3	c_3
b_4	c_4
b_5	c_5

=

A	B	C
a_1	b_1	*null*
a_2	b_2	*null*
a_3	b_3	c_3

In the following the set of syntactically correct Relational Algebra (*RA*) expressions or queries is defined recursively and the resulting schema of each expression is given.

1) For every relation $R \in \mathcal{R}$, R is an *RA* expression with schema *sch*(R).
2) A relation constant $\{(A_1 : d_1, \ldots, A_n : d_n)\}$ is an *RA* expression if $d_i \in val(D_i)$, $i = \overline{1,n}$. The schema of this expression is $(A_1 : D_1, \ldots, A_n : D_n)$.

Let E_{RA} be an *RA* expression with schema $RS = (A_1 : D_1, \ldots, A_n : D_n)$.

3) $\sigma_{A_i = A_j}(E_{RA})$, with $i, j \in \{1, \ldots, n\}$ is an *RA* expression with schema *RS*.
4) $\sigma_{A_i = d}(E_{RA})$, with $i \in \{1, \ldots, n\}$ and $d \in val(D_i)$ is an *RA* expression with schema *RS*.
5) $\pi_{B_1 \leftarrow A_{i_1}, \ldots, B_m \leftarrow A_{i_m}}(E_{RA})$ for $i_1, \ldots, i_m \in \{1, \ldots, n\}$ and $B_1, \ldots, B_m \in \mathcal{A}$ such that $B_j \neq B_k$ for $j \neq k$ is an *RA* expression with schema $(B_1 : D_{i_1}, \ldots, B_m : D_{i_m})$.

7. USING RELATIONAL ALGEBRA IN MODEL CHECKING ALGORITHM

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 $(b,e) \in U \Leftrightarrow \exists \langle j_1,\ldots,j_k\rangle \in \underline{D}(b)$ such as $\delta(b,j_1,\ldots,j_k) = e$. The labelling function for the graph G_S is defined as follows: $L: U \to \overline{M}$, $\forall\, u = (b,e) \in U$, $L(u) = \langle j_1,\ldots,j_k\rangle$, where $\delta(b,j_1,\ldots,j_k) = e$.

We define the relation schema (B:Q_B, M_1:D_1, …, M_k:D_k, E:Q_E) where $Q_B = \{b \in Q \mid \exists\, e \in Q$ such as $(b, e) \in U\}$, $Q_E = \{e \in Q \mid \exists\, b \in Q$ such as $(b, e) \in U\}$ and D_i, $i \in \{1, \ldots, k\} = \Lambda$ was defined in (1), such as if R_S is a relation name with schema defined above, (B:b, M_1:j_1, …, M_k:j_k, E:e) $\in R_S \Leftrightarrow \langle j_1,\ldots,j_k\rangle = L((b,e))$.

For a set $\mathcal{A}$ of m agents, $\mathcal{A} \subseteq \Lambda$, $\mathcal{A} = \{i_1, \ldots, i_m\}$, we define:

$$R_S(\mathcal{A}) = \pi_{B, M_{i_1}, \ldots, M_{i_m}, E}(R_S) \text{ where } i_l \in \mathcal{A}, l \in \{1, \ldots, m\} \text{ and}$$

$R_L(\mathcal{A}) = \pi_{B,LABEL \leftarrow M_{i_1} \circ \ldots \circ M_{i_m}, E}(R_S(\mathcal{A}))$ where the operator ° can be defined as follows: *i* ° *j* = *i* || ',' || *j*.

For a set $\Theta \subseteq Q_E$, $b \in Pre(\mathcal{A}, \Theta) \Leftrightarrow \exists\, j_{i_l} \in d_{i_l}(b), i_l \in \mathcal{A}, l = \overline{1,m}$ and $\exists\, e \in \Theta$ such as

$$(b, j_{i_1}, \ldots, j_{i_m}, e) \in R_S(\mathcal{A})$$

and $\nexists e' \in Q_E \setminus \Theta$ such as

$$(b, j_{i_1}, \ldots, j_{i_m}, e') \in R_S(\mathcal{A})$$

With other words, $b \in Pre(\mathcal{A}, \Theta) \Leftrightarrow \exists\ j_{i_l} \in d_{i_l}(b), i_l \in \mathcal{A}, l = \overline{1,m}$ such as

$$\pi_E(B : b, M_{i_1} : j_{i_1}, \ldots, M_{i_m} : j_{i_m}, E : Q_E) = \{(E : e) \mid e \in \Theta\}$$

In the following, the set of states $Q_E \setminus \Theta$ is denoted by $\overline{Q}$.

Now we can design an algorithm to compute the function *Pre*($\mathcal{A}$, Θ) using *RA* expressions:

Algorithm 2. Computing $Pre(A, \mathrm{Q})$ function using relational algebra expressions

Step1

$$\pi_{B,LABEL}(\sigma_{E \in \Theta}(R_L(\mathcal{A}))) = R_L^{\Theta}(\mathcal{A})$$
$$\pi_{B,LABEL}(\sigma_{E \in \overline{\Theta}}(R_L(\mathcal{A}))) = R_L^{\overline{\Theta}}(\mathcal{A})$$

Step2

$$\rho_x(R_L^{\Theta}(\mathcal{A})) \underset{x.B = y.B \wedge x.LABEL = y.LABEL}{\rtimes} \rho_y(R_L^{\overline{\Theta}}(\mathcal{A})) = R_L^{\Theta,\overline{\Theta}}(\mathcal{A})$$

Step 3

$$\sigma_{y.LABEL=null}(\pi_{x.B, y.LABEL}(R_L^{\Theta,\overline{\Theta}}(\mathcal{A}))) = R_L^{\Theta,\ null}(\mathcal{A})$$

Step 4

$$Pre(\mathcal{A}, \Theta) = \pi_{x.B}(R_L^{\Theta,null}(\mathcal{A}))$$

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

Algorithm 3. Computing $Pre(\mathcal{A}, \mathrm{Q})$ function using SQL statements

```
select distinct B from
(
  select distinct x.B, y.LABEL from
  (
    select distinct B, LABEL from model
    where E in Θ
  ) x
  left join
  (
    select distinct B, LABEL from model
    where E not in Θ
  ) y
  on x.B = y.B and x.LABEL = y.LABEL
  where y.LABEL is null
) z
```

We tested the above algorithm on three database servers: MySql, H2 and Microsoft SQL Server. Surprising, the MySQL and H2 deal much better with queries having the clause "IN" based on many values. Rewriting the above query using temporary tables to hold elements of the set Θ, was obtained a significant performance increase in all cases.

8. SOME EXAMPLES

In [19] is presented a CTL model for two processes competing for entrance into a critical section.

In the following, we present an ATL model for the critical section problem solved using a mutex. Our solution improves the mentioned CTL model because it supports true concurrency: the two processes can request simultaneously entrance into critical section, and their access is restricted using a mutex managed by the operating system (represented in our model by an agent).

If we consider our model presented in Figure 1 as a *concurrent game structure* $S=\langle\Lambda,Q,\Gamma,\gamma,M,d,\delta\rangle$, we will detail the semantics for the symbols from Γ - the set of propositions (labels from nodes representing states) and M – the set of agents moves. We have $\Gamma = \{I_1, I_2, W_1, W_2, E_1, E_2, L_1, L_2, F\}$ with the following significations:

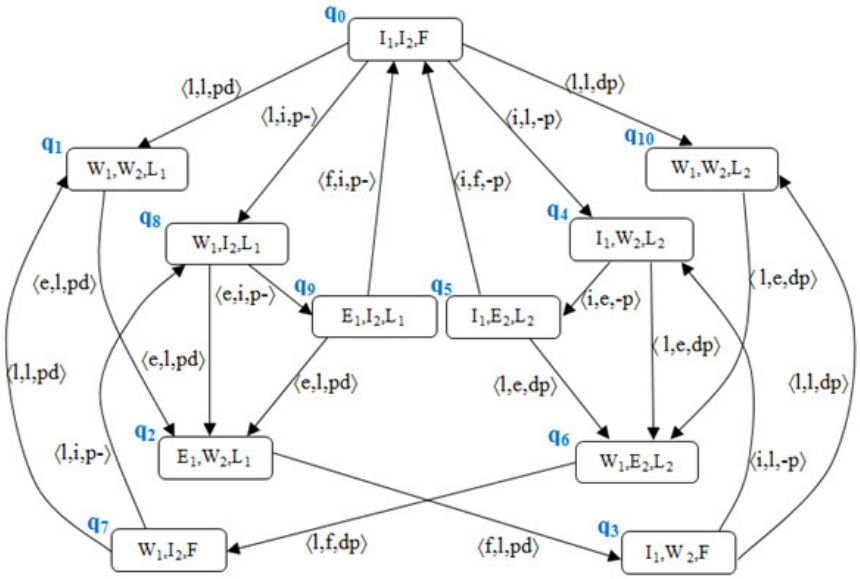

FIGURE 1: ATL model for two processes competing to entrance into a critical section.

- I_i – the process i is in ***I**dle* state, $i = \overline{1,2}$;
- W_i – the process i is in ***W**aiting* state (it is waiting to enter in critical section), $i = \overline{1,2}$;
- E_i – the process i is in ***E**xecuting* state (it is executing the code from critical section), $i = \overline{1,2}$;
- L_i – the mutex is owned (***L**ocked*) by the process i, $i = \overline{1,2}$;
- F – the mutex is not owned by any process (it has ***F**reed*).

The symbols from the set $M = \{l, e, i, f\} \cup \{pd, dp, p\text{-}, \text{-}p\}$ have the following significations:

- l – a request to enter in critical section (***l**ock* the mutex);
- e – a request to ***e**xecute* code from the critical section;
- i – there is no a request (***i**dle*);
- f – release (***f**ree*) the mutex, leave the critical section;
- pd – ***p**ermission* for agent 1, ***d**eny* for agent 2;
- dp – ***p**ermission* for agent 2, ***d**eny* for agent 1;

- *p-* – ***p****ermission* for agent 1, the agent 2 is *idle* (no request);
- *-p* – ***p****ermission* for agent 2, the agent 1 is *idle* (no request).

Using our model checking tool, we have verified that the following ATL formulas are satisfied by the model presented above:

ATL formula	Signification
not(<<𝒜>>~(E_1 *and* E_2))	*Safety* – Processes are not running simultaneously statements from the critical section
W_i => *not* (<<𝒜>># (*not* E_i)), $i = \overline{1,2}$	*Warranty* - each time one process tries to enter in critical section (owning the mutex), in the future it will succeed.
not (<<𝒜>>~ (*not* (I_i => << 𝒜>>@ W_i))), $i = \overline{1,2}$	*Nonblocking* – each process can require any time to enter in the critical section
<<𝒜>>~ (E_1 *and* (<<𝒜>> E_1 *U* (*not* E_1 *and* (<<𝒜>> *not* E_2 *U* E_1)))) <<𝒜>>~ (E_2 *and* (<<𝒜>> E_2 *U* (*not* E_2 *and* (<<𝒜>> *not* E_1 *U* E_2))))	*Without imposed succession* – the processes do not have the restriction to enter alternating in the critical section
E_i => L_i, $i = \overline{1,2}$	*Owning the mutex* – One process can execute the critical section only if it is owning the mutex
not(<<𝒜>>~ (*not* ((L_1 *or* L_2) => *not* (<<𝒜>># (*not F*)))))	*Releasing the mutex* – If one of the processes is owning the mutex, in the future it must release (free) the mutex
I_1 *and* I_2 => << 𝒜>>@ (W_1 *and* W_2)	*Concurrency* – If there is no process into critical section, both processes can request simultaneously to enter in the critical section, without blocking.

TABLE 1: ATL Formulas Satisfied By Our Model.

In the following we will apply the Algorithm 3 for computing function *Pre()* with different arguments passed in the process of checking of two ATL formulas from Table 1.

Example 1
For the ATL model presented above, we check the following ATL formula:

$$W_1 \Rightarrow \textit{not}\ (\langle\langle\mathcal{A}\rangle\rangle\#\ (\textit{not}\ E_1)) \qquad (2)$$

with its signification described in Table 1. The model checking algorithm will require some calls of function *Pre()* with certain arguments. In Table 2 are presented two computations of function *Pre()*:

	Q = {0,3,4,5,6,7,10}	
	$\mathcal{A}$={1}	$\mathcal{A}$={2}
$p_{x.B,y.LABEL}R_L^{Q,\bar{Q}}(\mathcal{A})$	B | LABEL 0 | NULL 0 | l 2 | NULL 3 | NULL 4 | NULL 5 | NULL 6 | NULL 9 | NULL 10 | NULL	B | LABEL 0 | l 2 | NULL 3 | NULL 4 | NULL 5 | NULL 6 | NULL 9 | NULL 10 | NULL
$Pre(\mathcal{A}, Q)$	{0,2,3,4,5,6,9,10}	{2, 3, 4, 5, 6, 9, 10}

TABLE 2: Computations of function $Pre(\mathcal{A}, Q)$ when checking the ATL formula (2).

For $\mathcal{A}$ = {1} because $i \in d_1(0)$, $\pi_E(B:0, M_1:i, E:Q_E) = \{(E:4)\}$, and $4 \in \Theta \Rightarrow 0 \in$ *Pre*($\mathcal{A}$, Θ).

For $\mathcal{A}$ = {2}, $d_2(0)$ = {*l*, *i*}. We have $\pi_E(B:0, M_2:i, E:Q_E) = \{(E:8)\}$, but $8 \notin \Theta$.

Also, $\pi_E(B:0, M_2:l, E:Q_E) = \{(E:1),(E:4),(E:10)\}$, but $1 \notin \Theta$.

We conclude that $0 \notin$ *Pre*($\mathcal{A}$, Θ).

Example 2

For the same ATL model, described in Figure 1, we consider the following formula:

$$not\ (<<\mathcal{A}>>\sim\ (not\ (I_1 => <<\mathcal{A}>>@\ W_1))) \quad (3)$$

with its signification also described in Table 1. In Table 3 are presented computations of function *Pre()* needed for checking the ATL formula (3):

	Q = {1,6,7,8,10}	
	A={1}	A={2}
$p_{x.B,y.LABEL}R_L^{Q,\bar{Q}}(\mathcal{A})$	B | LABEL 0 | NULL 3 | NULL 4 | NULL 5 | NULL 6 | NULL 7 | NULL 10 | NULL	B | LABEL 0 | l 0 | NULL 3 | l 4 | e 5 | NULL 6 | NULL 7 | NULL 10 | NULL
$Pre(\mathcal{A}, Q)$	{0,3,4,5,6,7,10}	{0,5,6,7,10}

TABLE 3: Computations of function $Pre(\mathcal{A}, Q)$ when checking the ATL formula (3).

For $\mathcal{A}$ = {2}, $d_2(3)$ = {*l*}. We have $\pi_E(B:3, M_2:l, E:Q_E) = \{(E:4),(E:10)\}$, but $4 \notin \Theta$.

Also, we have $d_2(4) = \{e\}$, and $\pi_E(B:4, M_2:e, E:Q_E) = \{(E:5),(E:6)\}$, but $5 \notin \Theta$. Thus, $3 \notin Pre(\mathcal{A}, \Theta)$ and $4 \notin Pre(\mathcal{A}, \Theta)$.

9. PUBLISHING THE ATL MODEL CHECKER AS A WEB SERVICE

Web services represent a standardized way for applications to expose their functionality over the Internet/Intranet, regardless of the platform or operating system upon which the service or the client is implemented. A Web service is accessible on the Web through an URL, and use a XML file, written using Web Service Definition Language (WSDL), to define its interfaces and bindings.

We choose to publish our implementation of the ATL model checker as a Web service in order to make the core of our tool accessible to various clients.

Our implementation is based on GlassFish/Tomcat as a Web container, and relies on

- MySQL
- SQLServer
- H2

as a database server.

For testing purposes, the ATL model checker described in this paper is available online via a Web service hosted by *mcheck-useit.rhcloud.com*.

The Web service will receive from a client the XML representation of a ATL model *S* and a ATL formula φ. The original form of the ATL model *S* is passed then to the algebraic compiler $\mathcal{C}$ generated by ANTLR using our ATL extended grammar. For a given ATL model (encoded as a directed multi-graph described in section 7) and an ATL formula φ, the Web service will parse the formula and will return to client the set of states in which the formula is satisfied if formula is syntactically correct, or a message describing the error from an erroneous formula.

In order to notify the client about possible syntactical errors found in the verified ATL formula, we must override the default behavior of the ANTLR error-handling. A custom error-handling in lexer is installed as follows:

```
@lexer::members {
   @Override
   public void reportError(RecognitionException re) {
     throw new RuntimeException("Lexical error!\n\n" +
                        "Position:" + re.line + ":" + re.charPositionInLine +
                        " erroneous character: '" + (char)re.c + "'");
       }
}
```

A syntactical error is reported if we install our handler in parser:

```
@members {
        @Override
        public void reportError(RecognitionException re) {
           throw new RuntimeException("Syntactical error!");
}
```

Finally, in case of occurrence of an error, we instruct ANTLR to throw that error, allowing the Web service to send it to the client:

```
@rulecatch {
       catch (RecognitionException err) { throw err; }
}
```

Our model checking tool is based on a C# GUI client who allows interactive graphical development of the ATL models. For internal representation of an ATL model as a directed multi-graph, our implementation is based on data structures provided by [20].

Thus, an ATL model encoding is based on symmetrically stored forward and backward adjacency lists. This paradigm supports an edge-oriented way of handling graphs with multiple edges.

The functionality of the client part is accessible through a right-click contextual menu which allows: adding nodes, labelling nodes, deleting nodes, adding arcs, display nodes numbers, etc., as we can see from the Figure 2.

In Figure 2, the labels of edges are associated with move vectors of agents depicted in Figure 1, and can be assigned in the ATL Designer in a dedicated window.

An overview of the system architecture of the ATL checker tool presented in this paper is given by the UML package diagram depicted in the Figure 3.

The ATL model checker tool contains the following packages:

- The algebraic compiler (*ATL Compiler*) invoked through the Web Service (ATL Checker);
- The GUI client application used for interactive construction of the ATL models as directed multi-graphs (*ATL Designer*);
- The *ATL non-GUI model* package contains classes used for programmatic construction of huge ATL models.
  Are available ATL API Client libraries for Java and C#.
- The *XML API for ATL models* package contains classes needed to encode the ATL model into XML.
  It is based on our XSD schema for specification of the XML representation of an ATL model;
- The *ATL GUI Model* package is responsible with graphical representation of the ATL concurrent game structures represented as directed multi-graphs.

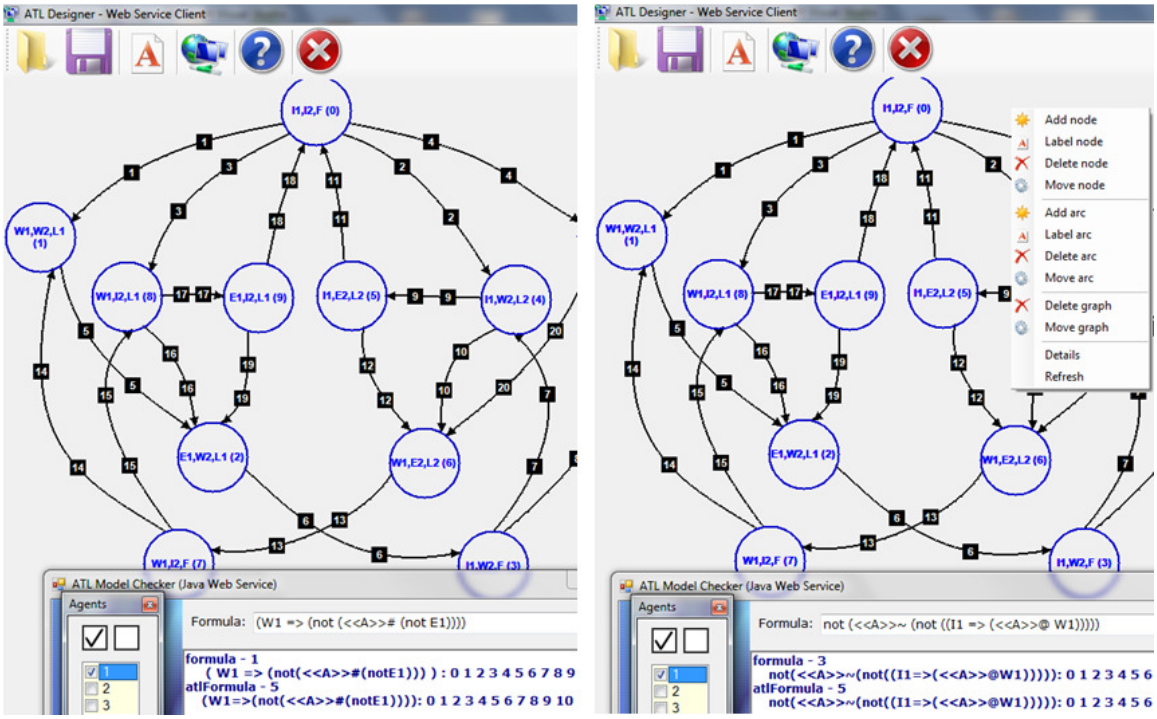

FIGURE 2: ATL Designer – Checking The ATL Formulas (2) and (3).

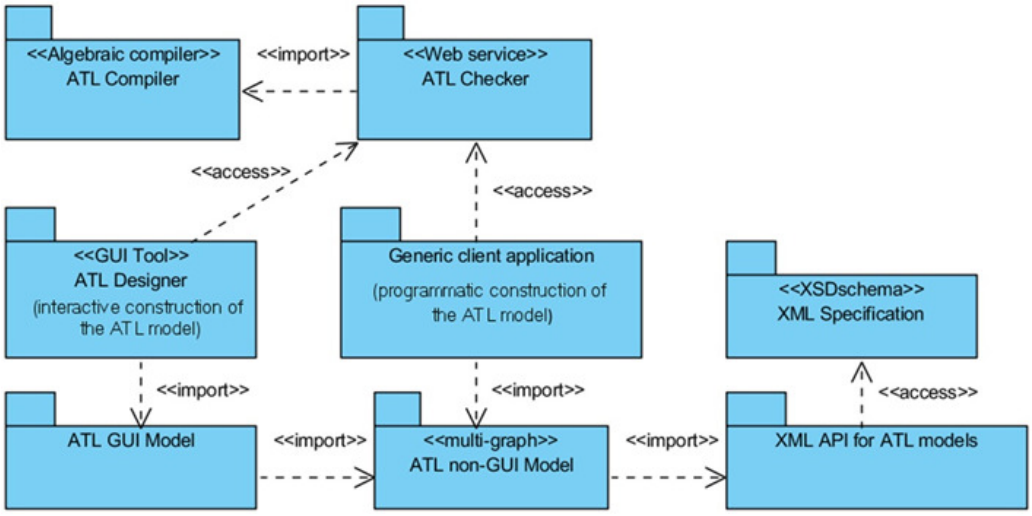

FIGURE 3: The System Architecture of The ATL Model Checker Tool.

All components of our ATL model checker tool can be downloaded from *http://use-it.ro*.

10. 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.

For the beginning we describe the usage of our model-checker to design a game strategy when playing Tic-Tac-Toe (called TTT for short in the rest of this paper). Although the game implemented is relatively simple, due to the large size of the structure representing the ATL model at the first moves, it represents a good opportunity to study the impact of technologies used to implement the model checker in its performance.

In [21] is showed that the model checking of computation tree logic (CTL) formulae can be used for generating plans in deterministic as well as non-deterministic domains. Because ATL is an extension of CTL that includes notions of agents, their abilities and strategies (conditional plans) explicitly in its models, ATL is better suited for planning, especially in multi-agent systems [1].

ATL models generalize turn-based transition trees from game theory and thus it is not difficult to encode a game in the formalism of concurrent game structures, by imposing that only one agent makes a move at any given time step.

The game TTT 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.

The implemented algorithm looks for infallible conditional plans to achieve a winning strategy that can be defined via ATL formulae.

We consider a computer program playing TTT game with a user (human) and the ATL model checking algorithm is used to return a strategy to achieve a winning strategy for the computer. The TTT is a turn-based synchronous game. In such a system, at every transition there is just one agent that is permitted to make a choice (and hence determine the future).

Formally, a game structure $S=\langle \Lambda, Q, \Gamma, \gamma, M, d, \delta \rangle$ is turn-based synchronous if for every state q from Q, there exist a player a from the set of all agents Λ such that $|d_b(q)| = 1$ for all players $b \in \Lambda \setminus \{a\}$. State q is the *turn* of player a.

In the following we will show how to use the ATL formalizations in finding winning strategies in case of TTT game.

10.1 Modelling The Game

We transform the original problem into an ATL model checking problem. More specifically, we want to determine a strategy $f_a : Q^+ \rightarrow D_a$ which leads the game into a winning state for the agent $a \in \Lambda$ representing the computer.
We suppose that positions of the board are numbered as in figure 4:

0	1	2
3	4	5
6	7	8

FIGURE 4: Labelling The Grids On The Board.

Formally, the turn-based synchronous game structure of TTT is defined as follows: $S=\langle \Lambda, Q, \Gamma, \gamma, M, d, \delta \rangle$.

The set of agents is $\Lambda = \{1,2\}$ and we consider that computer is represented by agent 1 and the user is represented by the agent 2.

Values of the board locations are denoted by $x_i \in \{0,1,2\}$, where $i \in \{0,1,...,8\}$. The value 0 means an empty position, the value 1 denotes a previous move of the agent 1 and the value 2 represents a move of the player 2. For the sequence of values $\overline{x_l x_m x_n}$ we define $\sum \overline{x_l x_m x_n} = \min(x_l,1) + \min(x_m,1) + \min(x_n,1)$ where $l,m,n \in \{0,1,...,8\}$.

The set of propositions (or observables) Γ is defined as follows:

$$\Gamma = \{(\overline{x_l x_{l+1} x_{l+2}}_{l=0,3,6}, \overline{x_l x_{l+3} x_{l+6}}_{l=0,1,2}, \overline{x_0 x_4 x_8}, \overline{x_2 x_4 x_6}, \overline{T}) \mid x_k \in \{0,1,2\} \text{ for } k = \overline{0,8} \text{ and } T \in \{1,2\}\}.$$

A state labelled with value $\overline{T} = 1$ signifies that is turn of the player 1 for making the move and if $\overline{T} = 2$ then the player 2 will make the next move.

The set of possible movements of agents is $M=\{0,1,2,3,4,5,6,7,8,9\}$.

For the agent 1, the set of alternative movements in the state $q \in Q$, in case when movements are still possible, is defined as:

$$d_1(q) = \begin{cases} \{1,...,k \mid k = 9 - \sum_{l=0,3,6} \overline{x_l x_{l+1} x_{l+2}} \geq 1,\ \overline{1} \in \pi(q)\} \\ \{0 \mid k = 9 - \sum_{l=0,3,6} \overline{x_l x_{l+1} x_{l+2}} \geq 1,\ \overline{2} \in \pi(q)\} \end{cases}$$

Analogue are defined the possible movements of the agent 2.

The game stops (so no moves are possible) if the board moves locations are full i.e.:

$$\sum_{l=0,3,6} \overline{x_l x_{l+1} x_{l+2}} = 9$$

Another situation where the game is not continuing is when a player won. The state q is a winning state for player 1 if $\overline{111} \in \gamma(q)$ and it is a winning state for player 2 if $\overline{222} \in \gamma(q)$.

Alternation to move can be formalized as follows: for a transition $\delta(q, j_1, j_2) = q'$, there are the following cases:

$$\bar{1} \in \gamma(q) \Rightarrow \bar{2} \in \gamma(q') \text{ or } \bar{2} \in \gamma(q) \Rightarrow \bar{1} \in \gamma(q')$$

In order to win the game, the player 1 (the computer) must follow two rules:

1. Try to choose at next move a state from the set ⟪1⟫ ◊ ($\overline{111}$), which favours the wining of the game in the future.
2. Avoid to choose at next move a state from the set ⟪2⟫ ○ ($\overline{222}$), to prevent the player 2 to win on the next move.

10.2 Experimental Results

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

In order to analyse their impact in the performance of the ATL model checker, were used three different database servers to implement the Web service, namely MySql 5.5, H2 1.3 and respectively Microsoft SQL Server 2008.

ATL-Designer permits the selection of one of the three database servers mentioned above:

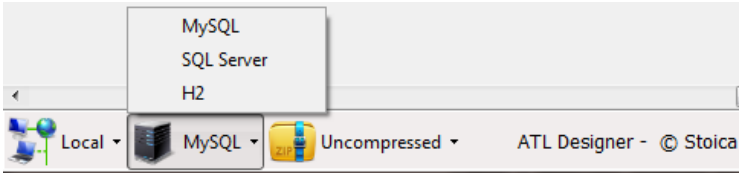

FIGURE 5: The Selection of The Database Server.

We have found important performance penalty due to the clause IN from the query presented in algorithm 2 from section 7, especially on Microsoft SQL Server 2008.

Thus the initial query was optimized by removing the clause IN and replacing it with JOIN operations performed between tables.

First of all, with states of the set Θ was built a temporary table in database server using the query:

Database server	Query syntax for building a table with states of the set Θ
SQL Server 2008	*insert into #Θ* *select distinct X.* from (values (q_1), (q_2), … (q_n)) as X(E)*
MySQL 5.5/ H2 1.3	*insert into `Θ` (E) values (q_1), (q_2), … (q_n)*

TABLE 4: Specific queries to populate tables with given discrete values.

where $q_i \in \Theta, i = \overline{1,n}$.

Then, to implement sub-queries were used temporary tables which have defined primary keys for fast access.

Supplementary optimizations were made for SQL Server:

- to reduce the transaction log and also to optimize insertions in tables of database *atl*, was used the directive:
  `alter database atl set RECOVERY SIMPLE`

- to optimize data transfer between the database server and Web server, was maximized the dimension of packets for network data transfer with the directive: **`EXEC sp_configure 'network packet size (B)', '32767'`**;

The MySQL server was configured only during the installation process.

The H2 database supports the in-memory mode (the data is not persisted), well suited for high performance operations. Also, H2 database can emulate the behavior of specific databases (DB2, Oracle, MySQL, PostgreSQL, etc.). Using MySQL Compatibility Mode made it possible to also use MySQL specific code / syntax for the H2 database.

Optimizations recommended in [22] are included in the following connection string for H2:

```
jdbc:h2:mem:db1;MODE=MySQL;LOG=0;LOCK_MODE=0;UNDO_LOG=0;DB_CLOSE_
DELAY=60
```

In Table 5 are presented the results showing the performance of our ATL model checker related to database server used:

Total time necessary to determine the winning strategy (Tic-Tac-Toe game) Intel Core I5, 2.5 GHz, 4Gb RAM			
Number of states	SQL Server 2008 (seconds)	MySQL 5.5 (seconds)	H2 1.3 (seconds)
4791	≈3.97	≈1.86	≈1.33
4255	≈3.37	≈1.62	≈1.17
3732	≈2.66	≈1.41	≈0.99
3423	≈2.32	≈1.24	≈0.90
3683	≈2.21	≈1.21	≈0.85
2307	≈1.97	≈0.86	≈0.58
2236	≈1.93	≈0.75	≈0.56

TABLE 5: A comparative analysis of impact of database servers in performance of ATL model checker.

In [23] is presented a comparison between Lurch (a random search model checker) and two well-known model checker tools, SMV and SPIN, showing the time and memory required, and the accuracy achieved by each tool when playing the tic-tac-toe game.

SPIN is a well-known explicit-state LTL (Linear Temporal Logic) model checker tool, and SMV is a symbolic CTL (Computation Tree Logic) model checker.

Although the logics LTL and CTL have their natural interpretation over the computations of closed systems and the logic ATL is used for the specification and verification of open systems, in theory the expressive power of ATL beyond CTL (in the case of closed systems ATL degenerates to CTL) comes at no cost - the model checking complexity of synchronous ATL is linear in the size of the system and the length of the formula [3].

Results from [23] showed that both SMV and SPIN were able to find an optimal strategy for a player in less than one second, on a 3x3 board.

As we can see from Table 5, the ATL model checker tool is not as fast as the CTL/LTL tools, but we must take into consideration that an ATL model is more expressive (with ATL we can quantify over the individual powers of one player or a cooperating team of players, ATL models capture various notions of synchronous and asynchronous interaction between open systems, etc.).

In [24] 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. [7]. Experimental results showed that the time necessary to find a winning strategy for a player, on a configuration with a Dural-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 2 seconds when H2 was used.

By using a database-based technology in the core of the ATL model checker, our tool provides a good foundation for further improvement of its performance and scalability.

In the actual stage of the development, experimental results are encouraging, showing that our tool is able to handle large systems efficiently.

11. CONCLUSIONS

In this paper we built an ATL model checking tool, based on robust technologies (Java, .NET, SQL) and well-known standards (XML, SOAP, HTTP). The implementation of the ATL model checking algorithm is based on Java code generated by ANTLR using an original ATL grammar and provides error-handling for eventual lexical/syntax errors in formula to be analysed. 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 XML interfaces.

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

Further investigation on improving performance will be done using in-memory databases (H2, HSQLDB). Also, we are planning to approach time constraints [25] in our ATL model checker. Because the SQL queries used in verification a composite ATL formula might consist of many subqueries that can be run in parallel, we would start looking at using the horizontal scalability features such as Parallel Pipelined Table Functions (PTF) provided by Oracle databases.

12. REFERENCES

[1] W. Van der Hoek, M. Wooldridge. “Cooperation, Knowledge, and Time: Alternating-time Temporal Epistemic Logic and its Applications”. *Studia Logica* 75, Kluwer Academic Publishers, pp. 125-157 2003.

[2] F. L. Cacovean. “Using CTL Model Checker for Verification of Domain Application Systems”, in Proceedings of the 11th International Conference on EVOLUTIONARY COMPUTING (EC '10), 13-15 Jun., Iasi, Romania, ISSN: 1790-2769, ISBN: 978-960-474-194-6, 2010, pp. 262-267.

[3] R. Alur, T. A. Henzinger, O. Kupferman. “Alternating-time Temporal Logic”. Journal of the ACM, vol. 49, pp. 672–713, 2002.

[4] M. Kacprzak, W. Penczek. “Fully symbolic Unbounded Model Checking for Alternating-time Temporal Logic”. Journal Autonomous Agents and Multi-Agent System, Volume 11 Issue 1, pp. 69 – 89, 2005.

[5] R.E. Bryant. “Graph-based algorithms for boolean function manipulation”. IEEE Transactions on Computers, C-35(8), pp. 677-691, 1986.

[6] K.Y. Rozier. “Survey: Linear Temporal Logic Symbolic Model Checking”. Computer Science Review, Volume 5 Issue 2, pp. 163-203, 2011.

[7] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, S. Tasiran. “Mocha: Modularity in Model Checking”, in Proceedings of the Tenth International Conference on Computer-Aided Verification (CAV), Lecture Notes in Computer Science 1427, Springer, 1998, pp. 521-525.

[8] Lomuscio, F. Raimondi. “MCMAS: A model checker for multi-agent systems”, in Proceedings of TACAS 06, volume 3920 of Lect. Notes in Computer Science, Springer-Verlag, 2006, pp. 450-454.

[9] C. Eisner, D. Peled. “Comparing Symbolic and Explicit Model Checking of a Software System”, in Proceedings SPIN Workshop on Model Checking of Software, volume 2318 of LNCS, Volume 55, pp. 230-239, 2002.

[10] G. Holzmann. “The model checker SPIN”. IEEE Trans. on Software Engineering, vol. 23, pp. 279–295, 1997.

[11] F. Lerda, N. Sinha, M. Theobald. “Symbolic Model Checking of Software”. Electronic Notes in Theoretical Computer Science, Volume 89, Issue 3, pp. 480–498, 2003.

[12] A.J. Hu. “Techniques for Efficient Formal Verification Using Binary Decision Diagrams”. PhD thesis, Stanford University. Internet: http://i.stanford.edu/pub/cstr/reports/cs/tr/95/1561/CS-TR-95-1561.pdf, 1995 [Mar. 29, 2016].

[13] B. Bingham, J. Bingham, F. M. De Paula, J. Erickson, G. Singh, M. Reitblatt. “Industrial Strength Distributed Explicit State Model Checking”, in Proceedings of the 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology (PDMC-HIBI '10). IEEE Computer Society, Washington, DC, USA, 2010, pp. 28-36.

[14] W. Jamroga, N. Bulling. “Comparing variants of strategic ability”, in Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume One, 2011, pp. 252-257.

[15] F. L. Stoica, M. F. Boian. “Algebraic approach to implementing an ATL model checker”. Studia Univ. Babes Bolyai, Informatica, Cluj-Napoca, Romania. Volume LVII, Number 2, pp. 73–82, 2012.

[16] T. Parr. The Definitive ANTLR Reference. Pragmatic Bookshelf, Dallas Texas, ISBN 0-9787392-5-6, 2007.

[17] W. Jamroga. “Easy Yet Hard: Model Checking Strategies of Agents”. Computational Logic in Multi-Agent Systems. Lecture Notes in Computer Science, Volume 5405, Springer Berlin Heidelberg, pp. 1-12, 2009.

[18] M. Herschel. “Introduction to Database Management Systems”. Internet: https://www.lri.fr/~herschel/courses_ws1314/resources/05_relational_algebra.pdf , 2013 [Mar. 29, 2016].

[19] T. Rus, E.V. Wyk, T. Halverson. “Generating Model Checkers from Algebraic Specifications”. Journal Formal Methods in System Design archive, Volume 20, Issue 3, pp. 249 – 284, 2002.

[20] J. Ebert. “A Versatile Data Structure for Edge-Oriented Graph Algorithms”. Communications of the ACM, Volume 30 Number 6, pp. 513-519, 1987.

[21] M. Pistore, P. Traverso. “Planning as model checking for extended goals in non-deterministic domains”, in Proceedings of IJCAI, 2001, pp. 479-486.

[22] H2 Database Engine, Internet: http://www.h2database.com/html/performance.html, 2014 [Mar. 29, 2016].

[23] D. Owen, T. Menzies. “Lurch: a Lightweight Alternative to Model Checking”, in Proc. Software Engineering and Knowledge Engineering (SEKE), 2003, pp. 158-165.

[24] J. Ruan. “Reasoning about Time, Action and Knowledge in Multi-Agent Systems”. Ph.D. Thesis, University of Liverpool. Internet: http://ac.jiruan.net/thesis/, 2008 [Mar. 29, 2016].

[25] F. L. Cacovean. “An Algebraic Specification for CTL with Time Constraints”, in Proc. First International Conference on Modelling and Development of Intelligent Systems, Oct. 22-25, Sibiu, Romania, ISSN 2067 – 3965, 2009, pp. 46-55.