"BABES-BOLYAI" UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS

Laura Florentina Cacovean

Formal verification of models for discrete event systems

THESIS SUMMARY

Scientific coordinator:
Prof. PhD. Florian Mircea Boian

2014

The thesis contains the following chapters [1]:

[1] In this summary are not detailed all sections of Chapters

Publications associated with doctoral thesis

The research results and original contributions presented in the doctoral thesis were published in proceedings of international conferences where I attended, in the Studia journal of "Babes Bolyai" University - Informatica or are under review at various ISI journals, classified by CNATDCU.

Published papers (3 conferences of category B, 1 conference of category C, 3 conferences of category D, 1 journal of category D):

1. **Laura Florentina Stoica**, Florian Mircea Boian and Florin Stoica. *A Distributed CTL Model Checker*. Proceeding of 10th International Conference on e-Business, Reykjavik Iceland, paper 33, pg. 379-386, 29-31 July, **2013**. Indexed in *ISI Thomson*. **Conference classified by CNATDCU in B category**. [SBS13] (http://www.bibsonomy.org/bibtex/2ec9b93632cd56d6c8daea8db853c41d9/dblp)
2. Florin Stoica and **Laura Florentina Stoica**. *Building a new CTL model checker using Web Services*. Proceeding The 21th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2013), At Split-Primosten, Croatia, 18-20 September, pg. 285-290, **2013.** ISBN: 978-1-4799-1122-6. DOI: 10.1109/SoftCOM.2013.6671858. Indexed in *ISI Thomson*. **Conference classified by CNATDCU in B category**. [SS13] (http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6671858)
3. **Laura Florentina Stoica**, Florin Stoica and Florian Mircea Boian. *Using ATL model checking in agent-based applications*. Proceeding of Third International Conference on Modelling and Development of Intelligent Systems, Sibiu, Romania, 10–12 October, pg. 127-135, **2013**. Indexată *Zentralblatt Math*. Anul publicării 2014. **Conference classified by CNATDCU in D category**. [SSB13]
4. **Laura Florentina Stoica** and Florian Mircea Boian. *Algebraic approach to implementing an ATL model checker*. STUDIA UNIV. BABEŞ BOLYAI, INFORMATICA, Cluj-Napoca, Romania. Volume LVII, Number 2, pg. 73–82, **2012. Journal classified by CNATDCU in D category**. [SB12] (http://www.studia.ubbcluj.ro/arhiva/cuprins.php?id_editie=710&serie=INFORMATICA&nr=2&an=2012)
5. **L.F. Stoica**, F. Stoica and D. Simian. *Client/Server Implementation of an ATL Model Checker Using Web Services*. Proceedings of the 16th WSEAS International Conference on Computers, Kos Island, Greece, pg. 359–364, July 14-17, **2012**. ISBN: 978-1-61804-109-8. [SSS12] (http://www.wseas.us/e-library/conferences/2012/Kos/COMCOM/COMCOM-00.pdf)
6. **L.F. Cacovean**, F. Stoica and D. Simian. *A New Model Checking Tool*. Proceedings of the European Computing Conference (ECC'11). Paris, France, pg. 358–364, April 28-30, **2011.** Indexată *Scopus*. **Conference classified by CNATDCU in C category.** [CSS11] (http://dl.acm.org/citation.cfm?id=1991016.1991081&coll=DL&dl=GUIDE&CFID=413406545&CFTOKEN=34793924)
7. **Laura Florentina Stoica** and Florin Stoica. *Considerations about the implementation of an ATL model checker*. Second International Conference on Modelling and Development of Intelligent Systems, MDIS. Sibiu, Romania, pg. 170–179, **2011**. Indexată *Zentralblatt Math*. **Conference classified by CNATDCU in D category.** [SS11] (http://conferences.ulbsibiu.ro/mdis/2011/Doc/Proceeding_mdis2011.pdf)
8. **Laura Florentina Cacovean** and Florin Stoica. *Modeling the Broker Behavior Using a BDI Agent*. Proceedings of the 14th WSEAS International Conference on Computers (CSCC). Corfu, Grecia, pg. 699–703, **2010**. Indexată *Scopus*. [CS10] (http://dl.acm.org/citation.cfm?id=1984366.1984415&coll=DL&dl=GUIDE&CFID=413406545&CFTOKEN=34793924)
9. F. Stoica and **L. F. Cacovean**. *Interoperability Issues in Accessing Databases through Web Services*. Proceedings of the 11th WSEAS International Conference on Evolutionary Computing (EC '10). Iaşi, Romania, pg. 279–284, **2010**. Indexată *Scopus*, *ISI Thomson*. [SC10]

(http://dl.acm.org/citation.cfm?id=1863431.1863478&coll=DL&dl=GUIDE&CFID=413406545&CFTOKEN=34793924)

10. **Laura Florentina Cacovean**. *Using CTL Model Checker for Verification of Domain Application Systems*, Proceedings of the 11th WSEAS International Conference on Evolutionary Computing (EC '10), 13-15 iunie **2010**, Iaşi, Romania, ISSN: 1790-2769, ISBN: 978-960-474-194-6. Indexată *ISI Thomson*. (http://dl.acm.org/citation.cfm?id=1863431.1863475&coll=DL&dl=GUIDE&CFID=413406545&CFTOKEN=34793924)

11. **Laura F. Cacovean** and Florin Stoica. *CTL Model Update Implementation Using ANTLR Tools*. Proceedings of the 13th WSEAS International Conference on COMPUTERS, Rhodos, Greece, pg. 169–174, **2009**. ISSN: 1790-5109, ISBN: 978-960-474-099-4. Indexată *ISI Thomson*. [CS09] (http://dl.acm.org/citation.cfm?id=1627733)

12. **Laura Florentina Cacovean**. *An Algebraic Specification for CTL with Time Constraints*. First International Conference on Modelling and Development of Intelligent Systems, MDIS'09, Sibiu, Romania, pg. 46–55, **2009**. ISSN 2067 - 3965.
Indexată *Zentralblatt Math*. **Conference classified by CNATDCU in D category.** [Ca09]
(http://www.zentralblatt-math.org/zbmath/search/?q=an%3A1240.65005)

13. **Laura F. Cacovean** and Florin Stoica. *Algebraic Specification Implementation for CTL Model Checker Using ANTLR Tools*. *WSEAS International Conferences, Computers and Simulation in Modern Science - Volume II*, Bucharest, Romania, pages 45–50, 2008. ISSN: 1790-5117, ISBN: 978-960-474-032-1. [CS08]
(http://www.wseas.us/e-library/conferences/2008/tomos2/papers/vol00.pdf)

14. **Laura Florentina Cacovean**, Marian Pompiliu Cristescu, Corina Ioana Cristescu, Ciprian Cucu. *Construction of a generalized model for determination the broker behaviour for capital market*. Proceedings of the 4th International Conference on Knowledge Management: Projects, Systems and Technologies Knowledge is power - KIP 2009 Bucureşti, 6-7 noiembrie **2009**, ISBN: 978-973-663-783-41.
(http://econpapers.repec.org/paper/romconfkm/17.htm)

15. **Laura F. Cacovean**, Iulian Pah, Emil M. Popa and Cristina I. Brumar. *Algorithm and an elevator control system example for the CTL model checker*. ICE-B 2008, International Conference on E-Business, Porto, Portugal, July 26-29, pg. 77–80, **2008**, ISBN: 978-989-8111-58-6.
Indexată *ISI Thomson*. **Conference classified by CNATDCU in B category.** [CPPB08]
(http://www.ice-b.icete.org/Abstracts/2008/ICE-B_2008_Abstracts.htm)

16. **L.F. Cacovean**, E.M. Popa, C.I. Brumar and I. Pah. *An application CTL formula based on Problem Solving Methodology*. New Aspects of Computers from Proceedings of the 12th WSEAS International Conference of Computers. Heraklion, Greece, pg. 218–223, **2008**. ISSN: 1790-5109, ISBN: 978-960-6766-85-5.
Indexată *ISI Thomson.* [CPBP08]
(http://dl.acm.org/citation.cfm?id=1513605.1513646&coll=DL&dl=GUIDE&CFID=413406545&CFTOKEN=34793924)

Papers under review in ISI journals classified by CNATDCU:

1. Florin Stoica and Laura Stoica. *Design, implementation and evaluation of a new ATL model checking tool*. Journal of Logical Methods in Computer Science. Nr: LMCS-2013-927. Decembrie 2013, ISSN 1860-5974 (http://www.lmcs-online.org/index.php), **Journal classified by CNATDCU in B category.**

2. Laura F. Stoica, Florin Stoica and Florian M. Boian. *Verification of JADE agents using ATL model checking*. International Journal of Computers Communications & Control. ID 803, 10.12.2013, ISSN 1841-9836 (http://journal.univagora.ro/), **Journal classified by CNATDCU in C category.**

1. Introduction

1.1. Actual approaches in verifying models

Testing and simulation can give us only confidence in the implementation of a software system, but cannot prove that all bugs have been found. However testing is neither exhaustive nor very effective for software, especially concurrent software, which is much more complex than sequential software. Thus, there has been a tremendous push for efficient algorithms and techniques that allow one to prove that a program satisfies certain properties. The process of stating and proving properties about programs is known as program verification.

Verification of a software system involves checking whether the system in question behaves as it was designed to behave. Design validation involves checking whether a system design satisfies the system requirements. Both of these tasks, system verification and design validation can be accomplished thoroughly and reliably using model-based formal methods, such as model checking [Roz11].

Model checking is particularly well-suited for the automated verification of finite-state systems, both for software and for hardware.

Main concern of formal methods in general, and model checking in particular, is helping to design correct systems [BBCR10]. Detecting and eliminating bugs as early in the design cycle as possible is clearly an economic imperative. For example, the Pentium FDIV bug (a bug in the Intel P5 Pentium floating point unit discovered in 1994) cost Intel Corporation a half billion dollars.

Model checking is the formal process through which a given specification representing a desired behavioral property is verified to hold for a given system (the model).

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), $\circ$ (at the next moment), $\Diamond$ (in future), $\Box$ (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 [SS11].

The branching time temporal logic CTL has a limited value when applied to open systems [HW02]. 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 [AHK02]. 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 $\langle\langle \mathcal{A} \rangle\rangle \varphi$ (where $\mathcal{A}$ is a group of agents). Informally, $\langle\langle \mathcal{A} \rangle\rangle \varphi$ means that agents $\mathcal{A}$ have a collective strategy to enforce φ, regardless of the actions of all the other agents [KP05].

The state explosion problem is widely agreed to be the most formidable challenge facing the application of model checking to large and complex real-world systems. In short, the number of states required by the model grows exponentially with the number of system components (or state variables), constituting the main practical limitation of model checking. Reducing the time required to verify models remains also a big challenge. This naturally raises interest in using parallelism to improve the performance of many formal model

checkers. Much of the extensive research on the parallelization of model checking algorithms followed the distributed memory programming model which appeared from the necessity to eliminate the memory constraints of a single computer system.

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

1.1.1. Remarkable Tools for CTL model verification

Some examples of well-known explicit-state parallel model checkers are DiViNe and PSPIN. DiVinE [BBCR10], [BBR10] is a distributed model checker for explicit state LTL (Linear Temporal Logic) model checking and is able to handle large systems consisting of as many as 419 million states, as stated in [BBPESR10]. PSPIN has also been used for performing distributed model checking with the capability of handling up to around 2.8 million states [LS99].

The basic idea behind symbolic model checking is to use a more efficient "symbolic" representation for the Kripke structure being checked and for sets of states of the Kripke structure. Since the sizes of these representations is typically the limiting factor in applying model checking, an efficient representation can potentially allow much larger structures to be checked.

Symbolic model checkers, such as CadenceSMV (Mir, 2000), NuSMV [CCGR02] analyse the state space symbolically using binary decision diagrams (BDDs). The binary decision diagram is a data structure for representing Boolean functions. With appropriate labelling of each state of the Kripke 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 [Roz11]. A symbolic model checker represents the Kripke structure itself symbolically using BDDs 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 Kripke structure explicitly.

1.1.2. Remarkable Tools for ATL model verification

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

In [AHMQRT98] 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 [AHMQRT98].

In [LR06] 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).

1.1.3. 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.

There are two measures of the size of the model under model checking. One is the number of the states in reachable state space (state space size), while the other is represented by the number of bits needed to represent

a state size. In symbolic model checking, the space used to represent a state is limited by the internal data structures. However, because a symbolic approach uses a compact representation of the set of states, it can handle a larger state space size. In explicit model checking, the state size is not strictly limited, but is related with state space size in the total memory consumption. In [EP02] is presented a comparison between RULEBASE, a symbolic model checker developed at IBM Haifa Research Laboratory and the explicit LTL model checker SPIN [Hol97]. 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 [EP02].

An important reason why software model checking is still predominantly performed using explicit-state model checkers such as SPIN, is that these methods gain much of their efficiency from state-reduction techniques such as partial-order reduction (POR). The POR methods explore a reduced state space in a property-preserving way [MP11]. Partial-order reduction is useful only when the system has an asynchronous model of composition.

Most hardware designs are based on a clocked-approach and thus are synchronous. For these systems, the symbolic model checking approach is more appropriate [LST03].

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 [Hu95]; this is because the communication mechanisms inherent in protocols tend to cause the BDDs in symbolic model checking to blow up [BBPESR10].

In their basic form, symbolic approaches tend to perform poorly on asynchronous models where concurrent interleaving are the main source of explosion, and explicit-state model-checkers with POR have been the preferred approach for such models [BBPESR10].

A detailed experimental comparison between performance of explicit-state model checkers and symbolic model checkers can be found in [Tab95]. The study follows an automata-theoretic approach in program verification, originally proposed by Vardi and Wolper [VW86]. Given a program *P* and a property φ, the task is to check whether the program satisfies the property φ. If the program *P* is viewed as a finite-state generator of words, and the specification φ as a finite-state acceptor, the model-checking problem is reduced to an automata-theoretic question: whether the automaton $A_P \cap A_{\neg\varphi}$ is empty. A non-deterministic automaton *A* defined over a nonempty finite alphabet Σ is said to be universal if it accepts Σ^*. The universality problem is to check if *A* is universal. If A_P is a universal automaton, the model checking problem is reduced to check whether A_φ is also universal. The study presented in [Tab95] uses two approaches, explicit and symbolic, for solving the universality problem. For evaluating the explicit approach, author has used Java tool Automaton.brics.dk [M∅l04] and the model-checker SPIN. To solve universality symbolically were used Cadence SMV [Mir00] and NuSMV [CCGPRST02] as symbolic BDD-based model checkers.

In order to present a full comparison between the symbolic and the explicit algorithms, was performed a scaling comparison of Cadence SMV, NuSMV and respectively SPIN. The direct comparison of the three model checkers shows that the explicit one (SPIN) is much faster than the symbolic ones. In conclusion, experimental results show that the explicit approach scales better than the symbolic one, which was rather surprising but confirms similar statements from [BBPESR10], [Hu95].

1.2. Motivation and objectives

The broad goal of our research was to develop a reliable, easy to maintain, scalable model checker tool to improve applicability of CTL (Computation Tree Logic) model checking in design of general-purpose computer software.

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 [Roz11]. 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.

To address the state explosion problem, our tool is based on an efficient data structure for internal representation of the model to be verified [GrStr12].

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 [BBPESR10]. Following this idea, our tool is based on Web Services technology to address the time constraints in verification of large models

1.3. Structure of the thesis

Chapter 1 presents the current state of research in using the temporal logics in verification of systems and an analysis of the advantages / disadvantages of explicit verification in relation to symbolic verification. Also, are analyzed briefly some dedicated tools for the CTL model checking, respectively ATL model checking. Further, is presented the motivation of the study undertaken, objectives and directions of research in the thesis.

In **chapter 2** are described the fundamental algebraic concepts used in the algebraic methodology of development of the algebraic compilers, in generally, and implementation of the model checkers, in particular.

Chapter 3 contains a comprehensive description of all aspects that must be considered in using the algebraic methodology to implement a software tool able to automatically verify systems modeled using temporal logic. This chapter begins with an overview of the concept of model checking based on temporal logic and the stages of the verification process of a model. In the following is described the algebraic methodology proposed by Rus [Rus91, CPBP08] for the design of compilers using algebraic specifications and their applications in the implementation of CTL model checkers. Section 3.2 includes the theory of algebraic compiler definition which is in fact a CTL model checker. In Section 3.3 is presented the abstract implementation level of a homomorphism between syntax algebra of source, respectively target languages. This homomorphism performs practically the checking of the CTL formulas in a given model. Section 3.4 contains the algebraic specification of a context-free language, which constitutes the premises of design the CTL model checker from context-free grammars which generate the language of the CTL formulas (process detailed in Section 3.4.2). Grammar specification of a $\sum$–language of CTL formulas is presented in section 3.5, and the chapter ends with the complete specification of algebraic compiler (CTL model checker). In the case study presented in Section 3.5.2, a CTL model for mutual exclusion of two processes, is showed the step by step execution of the designed algebraic compiler in the verification process of the modeled system, where specifications are expressed as CTL formulas. Although follows the same principles of the algebraic methodology proposed by Rus, our CTL model checker presents the following structural differences from the one shown in [Wyk98]: supports full syntax of CTL formulas, respectively all modal operators (in [Wyk98], CTL model checker supports only four temporal operators); differs the set of operations dependent on model, in our solution we opted for a set of operations that have a similar syntax used in ATL model checker, presented in chapter 6; algebraic compiler is generated using ANTLR, based on attribute grammars (described in chapter 4), and the parser is top-down, as opposed to the one provided by TICS tool used in [Wyk98], which is bottom-up and is based on macro processing in the translation process.

In **chapter 4** is shown the ANTLR parser generator, upon which is based the implementation of our CTL model checker and is justified the choice of this tool in the detriment of others (YACC, FLEX, BISON, BYACC/J, etc.). The attribute grammars are presented as an alternative for algebraic compiler development through macro processing (TICS system solution adopted, and applied in [Wyk98]) and are enumerated the arguments which recommended the ANTLR attribute grammars in implementing model checkers. Are

presented the ANTLR advanced concepts used in parser generation: LL(k), LL(*), PEG type analysis, flexible meta-language for specifying grammars, allowing placement of a semantic actions before and/or after specification of production rules, syntactic predicates, semantic predicates, *memoization*, finite automata decision with the role of prediction in the parsing, auto-backtracking for non-LL(*) grammars, techniques to eliminate nondeterminism. Also, is presented the technique of implementing the semantic actions in ANTLR, which is the concept of connection between attribute evaluation in the grammar that generates the language of CTL formulas and algebraic compiler implementation that represents the CTL model checker. Thus, the semantic action associated with a production rule represents the implementation of the derivate operation associated with the CTL operator for which it was defined that production. At the same time, the role of the semantic action is to compute the attribute value of the nonterminal which is rewritten by that production rule.

In **chapter 5**, for making available the CTL algebraic compiler implementation as reusable component of the CTL model checking tool, was achieved its publication as a Web Service. The CTL algebraic compiler, encapsulated in an Web service and based on the Java code generated by ANTLR on the basis of our original CTL attribute grammar, will perform the verification of CTL formulas in a given model, providing at the same time the signaling of any lexical/syntactic errors in the verified formula. The algorithm for determining a winning strategy for X0 game was used to evaluate the performance of the new CTL model checking tool.

In **chapter 6** is presented a CTL extension, named ATL (Alternating-time Temporal Logic). ATL temporal logic is used in modeling of open systems, and describe in a naturally way the processing of multi-agent systems, multi-user games, etc. Within the chapter it is shown how algebraic methodology can be used in the development of ATL model checking tool. In section 6.2 is formally defined the concurrent game structure. In section 6.3 is presented the ATL logic with its syntax and semantics. Sub-chapter 6.5 contains algebraic structure and description of an ATL model. Algebraic structure description of ATL language has as its starting point the definition of language of ATL formulas as Σ–language. The algebraic methodology to design a CTL model checker, presented in detail in chapter 3, has been successfully applied for the algebraic design of an ATL model checker. The ATL algebraic compiler specification was detailed in section 6.7, by defining the EBNF syntax and semantic actions corresponding to the production rules of context-free grammar which specify the Σ-language of ATL formulas. In sections 6.9 and 6.10 is accomplished the formalization of the *Pre()* function, used in all derivate operations corresponding to ATL modal operators, using concepts of Relational Algebra. A concrete call of this function (which in terminology of algebraic methodology is a model-dependent operation) was exemplified in the verification of specifications formulated in our original ATL model for the critical section problem solved using a mutex.

In **chapter 7** is presented the client / server architecture of the ATL model checker, which is based on the Web services technology to expose the functionality of its core component, the algebraic compiler developed in the previous chapter. The ATL model checker tool is composed by a client application which allows building of interactive ATL models, a server part (the Web service) and two API libraries (available for C# and Java languages) that allow the programmatically specification of an ATL model in XML format and checking the ATL formulas by invoking the Web service. Also, was evaluated the performance of our model checker tool in relation to three database servers: MySql, SQL Server and H2. Have been achieved two applications of ATL model checker tool, for determining the optimum strategies in multi-agent systems modeled as synchronous structures of alternative concurrent games and respectively for validating Finite State Machine behaviors of JADE agents.

Chapter 8 contains the conclusions of the thesis and presents directions for future research.

Within the doctoral thesis, tables and figures are numbered with consecutive numbers prefixed by the number of section in which they appear.

1.4. Original contributions of the thesis

The main original contributions of the thesis and the papers where they were published are:

- Expanding the syntax algebra Sin_{ctl} of L_{ctl} language presented in [Wyk98] with the set of operators $\{\rightarrow, AG, AF, EG, EF\}$. Construction of $T_{MC} : Sin_{ctl} \rightarrow Sin_M$ homomorphism was also extended by defining a derived operation $d_{MC}(op)$ in the word algebra Sin_M of target language for each new operator introduced [CS08, CS09, Ca09];
- Defining two model - dependent operations, $pre_\forall()$ and respectively $pre_\exists()$ which are used in all derived operations associated to CTL modal operators; this allowed simplification of the syntax of derived operations from Sin_M algebra [SBS13, SS13];
- Expanding the algebraic specification of a context-free language for the case in which the production rules of the grammar which generates that language are specified in EBNF syntax. This extension was necessary because ANTLR grammar for specification of the L_{ctl} Σ–language uses EBNF syntax for some production rules associated to CTL operators in order to eliminate the left recursion;
- Proof of obtaining denotation (the set of satisfaction) for $AG\,f$ CTL formula as fix point of $g(X) = [\![f]\!] \cap pre_\forall(X)$ function;
- The analysis of the complexity of CTL model checking algorithm;
- Implementation/generation of algebraic compiler through ANTLR attribute grammar for specification of the L_{ctl} Σ–language [CS08, CS11];
- Exposure of algebraic compiler functionality through a Web Service – *CTL Checker* [CSS11];
- *CTL Designer* – The client component of the CTL model checking tool, a GUI application developed in C# which allows the interactive construction and verification of CTL models [CSS11];
- Design an algorithm to determine the optimal strategy in an alternate synchronous game and its use for performance evaluation of the CTL model checker [SBS13, SS13];
- Defining the language of ATL formulas as *Σ–language*, definition of derived operations and algebraic compiler structure that represents the ATL model checker [SS11, SB12, SSS12];
- Implementation of an ATL algebraic compiler through ANTLR attribute grammar for specification of the L_{atl} Σ–language [SSS12];
- Building an alternate synchronous game structure (ATL model) for controlling access to a Web site [SSS12];
- Formalization of the *Pre*() function – considered model dependent operation in the syntax algebra of target language for the algebraic compiler – through relational algebra expressions and its implementation by translating the respective expressions in SQL language;
- Publication the ATL model checking tool as Web service – *ATL Checker* [SB12, SSS12];
- *ATL Designer* – The client component of the ATL model checking tool, a GUI application developed in C#, allows the construction and verification of the interactive of ATL models. For internal representation of an ATL model as an oriented multi-graph, our implementation is based on the data structure appropriate for dynamic graphs [Ebe87]. These structures have been adapted for C# and then extended for representation of the concurrent game structures [SB12, SSS12];

- Development of an API programming interface – ATL Library – for the programmatic building of CTL/ATL models with large size;
- Building and verification of ATL model for the two concurrent processes who want to enter into a critical section. Our solution improves the classical CTL model because supports real competition: two processes may require simultaneously entering in the critical section, and their access is restricted using a mutex managed by the operating system, represented in our model by an agent;
- Modifying the ATL model proposed by Alur [AHK02] to be able to represent the agents movements through arbitrary symbols (in the original model the agents moves were represented through natural numbers) [SSB13, SB12];
- Design an algorithm to determine the optimal strategy in an alternate synchronous game and its use for performance evaluation of the of ATL model checker;
- Develop a technique for validation of Finite State Machine behaviors of JADE agents. The proposed solution is based on Java version of ATL Library Component, and allows checking at execution time agent specifications expressed through ATL formulas. The ATL model is constructed automatically, at definition of the finite state machine of JADE agents;
- Current implementation of the ATL model checker supports three database servers: MySQL, SQL Server and H2.

1.5. Keywords

Software system, Formal verification, Temporal logic, Model checker, CTL logic, CTL model checking, Kripke structure, CTL model, ATL logic, ATL model checking, ATL model, Concurrent game structure, Multi-agent system, Algebraic compiler, ANTLR context-free grammar, ANTLR attribute grammar, Web Service, Relational Algebra, SQL.

2. Fundamental theoretical concepts

This chapter describes fundamental algebraic concepts used in algebraic methodology, in general for the algebraic compiler development, and in particular for the implementation of the model checkers.

In the algebraic methodology the languages are represented using sigma algebras, and a compiler is specified as a generalized homomorphism which includes the source language in the target language.

Exposed methodology can be used to implement model checkers as algebraic compilers in which source languages are the languages of temporal logics that define the syntax and semantics of temporal logic formulas, and the target language is the language of set of states that satisfy the logical formulas within specific models.

3. A CTL model checking tool based on the algebraic methodology

Temporal logic has emerged as a main formalism for reactive systems. A temporal logic is essentially an ordinary predicate or propositional logic with the addition of modal operators for describing how the interpretation of symbols changes over time [Hu95]. Typical temporal operators include the next-time operator (X), the eventuality operator (F), the always operator (G) and until (U) operator.

A CTL model checker is a tool which can be used to verify that a given system satisfies a given CTL logic formula. A CTL model is a Kripke structure represented by a directed graph where the nodes are the states of the system and the edges represents the state transitions. The nodes are labeled with atomic propositions. In order to be verified by a given model, a property is written as a temporal logic formula over the labeled

propositions from the model. A model checker is an algorithm that determines the states of a model that satisfy a temporal logic formula.

3.1. The CTL model

A model is defined as a Kripke structure $M=(S, Rel, P:S\rightarrow 2^{AP})$ where S is a finite sets of states also called nodes, $Rel \subseteq S \times S$ is a transition relation denoting a set of directed edges, and P is a labelling function that defines for each state $s \in S$ the set $P(s)$ of all atomic propositions from AP that are valid in s. The transition relation Rel is left-total, i.e., $\forall s \in S \; \exists s' \in S$ such that $(s,s') \in Rel$.

For each $s \in S$, the notation $succ(s)=\{s' \in S \mid (s,s') \in Rel\}$ is used to denote the set of successors of s. From definition of Rel, each state from S must have at least one successor, that is $\forall s \in S, succ(s) \neq \emptyset$. A path in M is an infinite sequence of states $(s_0,s_1, s_2,...)$ such that $\forall i, i \geq 0$, we have $(s_i,s_{i+1}) \in Rel$.

We use $s' \in succ(s)$ to denote that there is a relation (s, s') in Rel. The labelling function P maps for each state $s \in S$ the set $P(s)$ of all atomic propositions from AP that are valid in s [HR00].

3.2. CTL syntax and semantics

A CTL formula has the following syntax given in Backus-Naur Form (BNF):

$\varphi :: true|false|ap|(\neg \varphi_1)| \varphi_1 \wedge \varphi_2| \varphi_1 \vee \varphi_2| \varphi_1 \Rightarrow \varphi_2| AX \varphi_1| EX \varphi_1| AG \varphi_1| EG \varphi_1| AF \varphi_1| EF \varphi_1|\varphi_1 AU \varphi_2| \varphi_1 EU \varphi_2, \forall ap \in AP$.

A CTL specification is interpreted over Kripke structures. 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 (an initial 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. The CTL syntax include several operators for describing temporal properties of systems: *A* (for all paths), *E* (there is a path), *X* (at the next moment), *F* (in future), *G* (always) and *U* (until)

Syntactically, *CTL* formulas are divided into three categories:

- those whose outermost operator, if any, is not a temporal operator;
- those whose outermost operator is a temporal operator (*X* (next), *U* (until), *F* (eventually) or *G* (always)) prefixed with the existential path quantifier *E*, and
- those whose outermost operator is a temporal operator prefixed with the universal path quantifier *A*.

3.3. Fixed-point characterization of CTL

Let $M=(S, Rel, P:S\rightarrow 2^{AP})$ be an arbitrary finite Kripke structure. Given a state s in S, is defined a satisfaction relation $(M, s) \vDash \varphi$ to specify that formula φ holds in s. We denote by $[\![\varphi]\!]_M = \{s \in S \mid (M,s) \vDash \varphi\}$ the set of all states from S which satisfy the formula φ (the set of states at which φ is *true*). $[\![\varphi]\!]_M$ is called the denotation of φ in model M. Because often M is implicit, we write $[\![\varphi]\!]$ rather than $[\![\varphi]\!]_M$. Thus $(M, s) \vDash \varphi \Leftrightarrow s \in [\![\varphi]\!]$.

Let 2^S denote the power set of the set S. A set valued function $f : 2^S \rightarrow 2^S$ is called monotone if for all $X, Y \subseteq S, X \subseteq Y \Rightarrow f(X) \subseteq f(Y)$.

Fixed points definition Let $f : 2^S \rightarrow 2^S$ be a set valued function and $Z \subseteq$ S a subset of S.

1. Z is called a fixed point of f if $f(Z) = Z$.
2. Z is called the least fixed point (*LFP*) of f if $f(Z) = Z$ and $\forall\, U \subseteq S, f(U) = U \Rightarrow Z \subseteq U$.
3. Z is called the greatest fixed point (*GFP*) of f if $f(Z) = Z$ and $\forall\, U \subseteq S, f(U) = U \Rightarrow U \subseteq Z$.

The Kleene fixed-point theorem can be written in the following form:

Theorem: Let $f : 2^S \rightarrow 2^S$ be a monotone function on a finite set S.

1. There is a least and a greatest fixed point of *f*.
2. $\bigcup_{n\geq 1} f^n(\phi)$ is the least fixed point of *f*.
3. $\bigcap_{n\geq 1} f^n(S)$ is the greatest fixed point of *f*.

The universal and existential *pre-image* functions $pre_\forall$, $pre_\exists : 2^S \rightarrow 2^S$ are defined by:

$$pre_\forall(X) = \{s \in S \mid succ(s) \subseteq X\} \qquad (1)$$
$$pre_\exists(X) = \{s \in S \mid succ(s) \cap X \neq \phi\}$$

For a CTL formula φ, the model checker will compute $[\![\varphi]\!]$ recursivelly, using the rules described in the following table, where *LFP* and *GFP* represent the least fixed point and respectively the greatest fixed point of the specified functions:

Formula **φ**	Computation of $[\![\boldsymbol{\varphi}]\!]$
ap	$\{ s \in S \mid ap \in P(s) \}$
true (false)	$S\ (\phi)$
$\neg\varphi_1$	S \ $[\![\varphi_1]\!]$
$\varphi_1 \wedge \varphi_2$ $\varphi_1 \wedge \varphi_2$	$[\![\varphi_1]\!] \cap [\![\varphi_2]\!]$ $[\![\varphi_1]\!] \cup [\![\varphi_2]\!]$
$AX\ \varphi_1$	$\text{pre}_\forall([\![\varphi_1]\!])$
$EX\ \varphi_1$	$pre_\exists([\![\varphi_1]\!])$
$\varphi_1 AU\ \varphi_2$	*LFP* of $f(X) = [\![\varphi_2]\!] \cup ([\![\varphi_1]\!] \cap pre_\forall(X))$
$\varphi_1 EU\ \varphi_2$	*LFP* of $f(X) = [\![\varphi_2]\!] \cup ([\![\varphi_1]\!] \cap pre_\exists(X))$
$AG\ \varphi_1$	*GFP* of $f(X) = [\![\varphi_1]\!] \cap pre_\forall(X)$
$EG\ \varphi_1$	*GFP* of $f(X) = [\![\varphi_1]\!] \cap pre_\exists(X)$
$AF\ \varphi_1$	*LFP* of $f(X) = [\![\varphi_1]\!] \cup pre_\forall(X)$
$EF\ \varphi_1$	*LFP* of $f(X) = [\![\varphi_1]\!] \cup pre_\exists(X)$

Table 3.3.1: Recursively computation of denotation $[\![\varphi]\!]$ of CTL formula φ.

In the following we will justify the form of denotation for CTL formula $AG\ \varphi$. By definition, $[\![AG\ \varphi]\!] = \{s_0 \in S \mid \forall\, (s_0, s_1, \dots), s_i \in [\![\varphi]\!]\ \forall i \in \mathbb{N}\}$. Thus, we can rewrite:

$[\![AG\ \varphi]\!] = \{s \in S \mid s \in [\![\varphi]\!]\ and\ \forall\, s' \in succ(s),\ s' \in [\![AG\ \varphi]\!]\} = [\![\varphi]\!] \cap \{s \in S \mid succ(s) \subseteq [\![AG\ \varphi]\!]\}$
$= [\![\varphi]\!] \cap pre_\forall([\![AG\ \varphi]\!])$.

So, $[\![AG\ \varphi]\!]$ is a fixed point of the function $f(X) = [\![\varphi]\!] \cap pre_\forall(X)$. It remains to see that it is the greatest fixed point.

Let *H* be another fixed point, i.e., $= [\![\varphi]\!] \cap pre_\forall(H)$. We must show that $H \subseteq [\![AG\ \varphi]\!]$.

Suppose the contrary: there is a state $h_0 \in H$ such that $h_0 \notin [\![AG\ \varphi]\!] \Rightarrow \exists\, \pi_M = (h_0, h_1, \dots)$ a path in *M* and $\exists\, k \in \mathbb{N}$ such that $h_k \in \pi_M$ and $h_k \notin [\![\varphi]\!]$.

But $h_0 \in H \subseteq [\![\varphi]\!] \Rightarrow h_0 \in [\![\varphi]\!]$. Also $h_0 \in H \subseteq pre_\forall(H) \Rightarrow h_1 \in H$ because $h_1 \in succ(h_0)$. Following the same reasoning $h_1 \in H \subseteq [\![\varphi]\!] \Rightarrow h_1 \in [\![\varphi]\!]$. Having as induction hupothesis $h_{k-1} \in H$ this imply that:

$$h_{k-1} \in pre_\forall(H) \Rightarrow h_k \in H \Rightarrow h_k \in [\![\varphi]\!],$$

contradiction with the initial assumption.

In conclusion, we proved that $H \subseteq [\![AG\ \varphi]\!]$ and thus $[\![AG\ \varphi]\!]$ is the greatest fixed point of the function $f(X) = [\![\varphi]\!] \cap pre_\forall(X)$. The algorithm for effective computation of $[\![AG\ \varphi]\!]$ is presented in the following section.

4. CTL model checking by attribute grammars

The CTL model checker is provided as a compiler $\mathcal{C}: L_s \rightarrow L_t$, where L_s is the source language and L_t is the target language. The source language L_s is the language describing the CTL formulas and the target language L_t is a language which describes the set of nodes from the model M where the corresponding CTL formulas are satisfied.

The compiler $\mathcal{C}$ translates a formula φ of the CTL model to the set of nodes $[\![\varphi]\!]$ over which formula φ is satisfied. That is, $\mathcal{C}(\varphi) = [\![\varphi]\!]$ where $[\![\varphi]\!] = \{s \in S \mid (M,s) \models \varphi\}$.

The implementation of the compiler $\mathcal{C}$ is made in two steps. First, we need a syntactic parser to verify the syntactic correctness of a given formula φ. Then, we should deal with the semantics of the CTL language, respectively with the implementation of the CTL operators presented in table 3.3.1.

Writing a translator for certain language is difficult to be achieved, requiring time and a considerable effort [Rus91]. Currently there are specialized tools which generate most of necessary code beginning from a specification grammar of the source language.

For implementation of the algebraic compiler we choose the ANTLR (***An**other **T**ool for **L**anguage **R**ecognition*). ANTLR [Parr07] 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.

ANTLR support the EBNF (Extended BNF) notation, useful for specification of operations that requires the use of recursion.

In order to translate a formula φ of a CTL model to the set of nodes $[\![\varphi]\!]$ over which formula φ is satisfied, is necessary to attach actions to grammatical constructions within specification grammar of CTL.

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 CTL formula. In case of our compiler $\mathcal{C}$, the actions define the semantics of the CTL model checker, i.e., the implementation of the CTL operators.

The model checker generated by ANTLR from our specification grammar of CTL takes as input the model M (where are defined the sets S, Rel, and P) and a formula φ, and provides as output the denotation of φ – the set of states where the formula φ is satisfied, using the following general algorithm:

- assign atomic propositions by labelling function P;
- handle Boolean operators by standard set operations;
- handle temporal operators AX, EX by computing pre-images using expressions given in (1);
- handle temporal operators AG, EG, AF, EF, AU, EU by applying rules described in table 3.3.1, until a fixpoint is reached.

The algorithm for computing $[\![AG\ \varphi]\!]$ is presented in figure 4.1 [CGL96].

For the formal specification of the AG operator given in figure 4.1, the corresponding *action* included in our ANTLR grammar of CTL language is detailed in figure 4.2.

```
Z:=∅; Z':= [[φ]];
while (Z≠Z') do
   Z:=Z';
   Z':=Z'∩ pre∀(Z');
endwhile
[[AG φ]]:=Z';
```

Figure 4.1: Formal definition of the set expression $[\![AG\ \varphi]\!]$.

```
private HashSet PreAll(HashSet Z) {
  HashSet rez = new HashSet();
  for (Node n1 : model) {
    Iterator<Edge> it =
    n1.getLeavingEdgeIterator();
    HashSet succ = new HashSet();
    while (it.hasNext()) {
       Edge e = it.next();
       Node n2 = e.getTargetNode();
       succ.add(n2.getIndex());
    }
    if (Z.containsAll(succ)) {
       rez.add(n1.getIndex());
    }
  }
  return rez;
}

ctlFormula returns [HashSet set]
@init { }
: 'ag' e=implExpr {
    HashSet rez = new HashSet();
    HashSet rez1 = new HashSet($e.set);
    while (!rez.equals(rez1)) {
       rez.clear();
       rez.addAll(rez1);
         HashSet tmp = PreAll(rez1);
       rez1.retainAll(tmp);
    }
    $set = rez1;
}
```

Figure 4.2: Implementation of the AG operator in ANTLR.

Analog were implemented all CTL temporal operators.

For efficient representation of CTL models, our tool is based on SingleGraph class from GraphStream package [GrStr12].

5. The architecture of the CTL model checker tool. Applications. Experimental results.

Web services represent a standardized way for applications to communicate with other applications over a network, regardless of the platform or operating system upon which the service or the client is implemented. We choose to publish our implementation of CTL model checker as a Web service in order to utilize the combined resources of distributed computers and to bring advantages of distributed verification to various clients over the Web. As we can see from the figure 5.1, the transport protocol (HTTP) used by the Web Service enables clients to invoke its methods through firewalls.

The architecture of the Web service implementation is represented in figure 5.1.

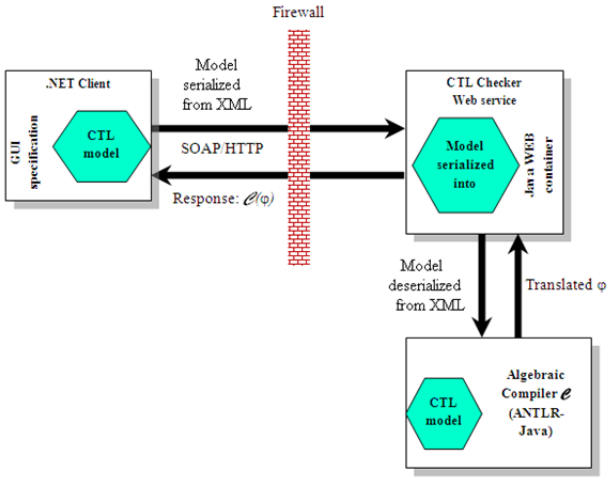

Figure 5.1: The architecture of the ATL model checker Web Service.

The Web service will receive from a client the XML representation of a CTL model S and a CTL formula φ to be verified. The original form of the CTL model S is then reconstructed and passed to the algebraic compiler $\mathcal{E}$ generated by ANTLR using our CTL extended grammar. For a syntactically correct formula φ, the compiler will return as result $\mathcal{E}(\varphi)=\{q\in Q \mid q \vDash \varphi\}$, the set of states in which the formula is satisfied. If as input is an erroneous formula φ, the model checker will return to client an message describing the error. In our tool is enabled a compression facility for large CTL models, to reduce the network traffic between client and server.

Our Web service is using GlassFish or Tomcat as a Web container. For testing purposes, the CTL model checker described in this paper is available online via two Web services hosted by *use-it.ro* and respectively by *mcheck-useit.rhcloud.com*.

The system architecture of the CTL checker tool presented in this paper is depicted in the following UML package diagram:

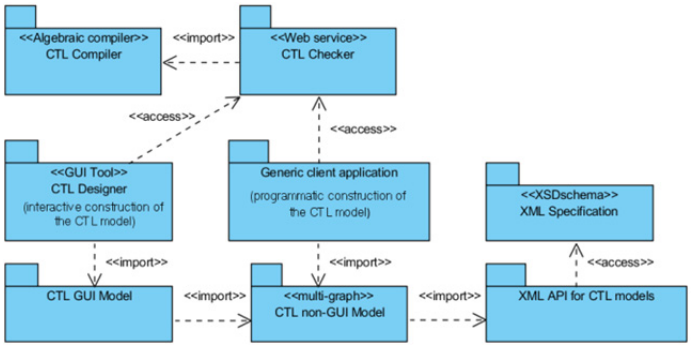

Figure 5.2: The system architecture of the CTL model checker tool.

The CTL model checker tool contains the following packages:

- The algebraic compiler (CTL Compiler) embedded into the Web Service (CTL Checker); implementation of these components was made in Java.
- The GUI client application written in C# and used for interactive construction of the CTL models as directed graphs (CTL Designer).
- In case of huge CTL models, with many states, is required the use a programmatic construction of these models. The *CTL non-GUI model* package contains classes used for internal representation of a CTL model as a directed graph. There are available two libraries, for C# and respectively for Java. For internal representation of a CTL model in C#, our implementation is based on data structures

provided by (Ebert, 1987), more precisely symmetrically stored forward and backward adjacency lists. The Java implementation is based on GraphStream [GrStr12].

- The *XML API for CTL models* package contains classes needed to encode the CTL model into XML.
- The *CTL GUI Model* package is responsible with graphical representation of the Kripke structures as directed graphs.

The model checking tool contains a C# GUI client who allows interactive graphical design of the CTL models.

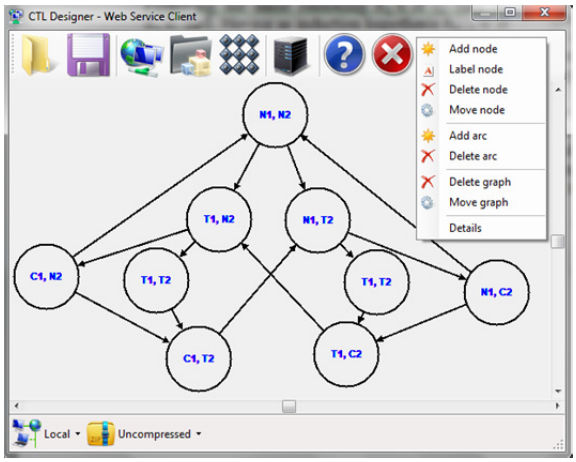

Figure 5.3: CTL Designer - the C# client in action.

The model is sent as a XML document to the Web service, together with the formula to be verified. The client is also responsible for displaying the response from server.

All facilities related to interactive design of CTL models are accessible through a right-click contextual menu: adding nodes, labelling nodes, deleting nodes, adding arcs, display nodes numbers, etc.

In addition, the CTL Designer interface allows several configurations:

Icon	Signification
	Allows selection of the Web server: local or remote (Internet). Service-location details for web-service access can be found at: http://use-it.ro
	Allows selection of the implicit Internet server.
	In case of huge models, is recommended to activate their compression before sending them to the Web Service.

Table 5.1: CTL Designer configuration from its interface.

5.1. Performance evaluation of our tool

In this section 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).

First, let us describe the (classical) game of Tic-Tac-Toe. The game is played by two opponents, *X* and *O*, 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. If the entire board becomes full but no player has formed a line, the result is a draw.

In our example, player *X* is played by the application and player *O* should be played by a human.

CTL 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).

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

Modelling the Game

We suppose that positions of the board are numbered as in figure 5.1.1:

0	1	2
3	4	5
6	7	8

Figure 5.1.1: Labelling the grids on the board

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 player *X* and the value 2 represents a move of the player *O*.

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 $l,m,n \in \{0,1,...,8\}$.

Formally, the Kripke model of *TTT* is defined as $M = (S, Rel, P{:}S \rightarrow 2^{AP})$ with its structure explained in the following.

The set of atomic propositions *AP* is denoted by:

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

The number of successors of a state is given by the formula:

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

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

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 *s* is a winning state for player *X* if $\overline{111} \in P(s)$ and it is a winning state for player *O* if $\overline{222} \in P(s)$.

Alternation to move can be formalized as follows: for a transition $(s,s') \in Rel$, there are the following cases:

$$\overline{T} \in P(s) \Rightarrow \overline{3-T} \in P(s')$$

where $T \in \{1,2\}$.

Algorithm to determine the optimal strategy

Assuming that the game is in the state $s_0 \in S$, we denote by *k* the number of empty positions of the board. The strategy of player *X* can be expressed by the following algorithm:

Step 1	Determines all states from the model satisfying the formula: $(\mathrm{AX\ EX})^{k/2}(\mathrm{AX}\ \overline{111})$, to choose the move which favours wining of the game in the future. We denote this set with WIN1.

Step 2	Determines all states from the model satisfying the formula: EX $\overline{222}$, to prevent player 2 to win on the next move. We denote this set with WIN2.

Step 3	If (WIN1\WIN2) $\cap$ succ(s_0) $\neq \varnothing$ then
	Choose randomly a state s from the resulting set.
	Else If succ(s_0)\WIN2 $\neq \varnothing$ then
	Choose randomly a state s from the resulting set.
	Else
	Choose randomly a state s from the set succ(s_0).
	End If
	Set s as current state.

Step 4	If $\overline{111} \in P(s)$ then STOP. The player X has won. If the board is full is declared equality and STOP.
Step 5	Player O performs moving. If $\overline{222} \in P(s)$ then STOP. The player O has won. If the board is full is declared equality and STOP else go to step 1.

In the following we present a game scenario implemented using the CTL model checker API.

At first move, the computer (player X) chooses the position 0. After the player O moves, is constructed the CTL model of the game. This model has 2307 states and 3330 transitions.

In figure 5.1.2 can be seen that player X has determined three winning strategies. It chooses randomly one from them and follows it performing the corresponding move.

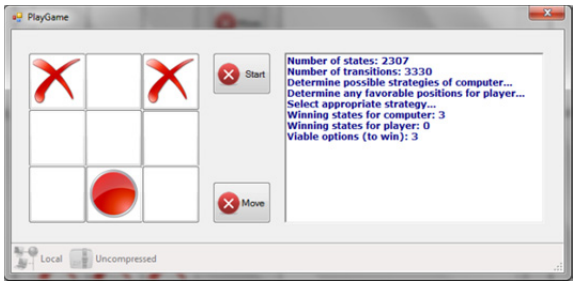

Figure 5.1.2: The move of player X (in position 2) which follows a winning strategy

Finally, can be seen that player O could not avoid defeat, because the player X follows a winning strategy:

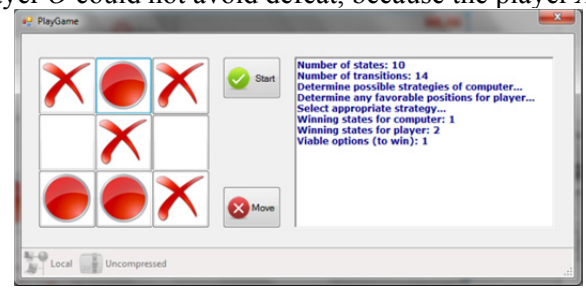

Figure 5.1.3: The player X (computer) won

Experimental results

Although the game implemented is relatively simple, due to the large size of the structure representing the CTL model at the first moves, it represents a good opportunity to study the effectiveness of our approach in designing and implementing a CTL model checker.

In the figure 5.1.4 are presented the results showing the performance of the CTL model checker when running on Intel Core I5, 2.5 GHz, 4Gb RAM to find a winning strategy for Tic-Tac-Toe game.

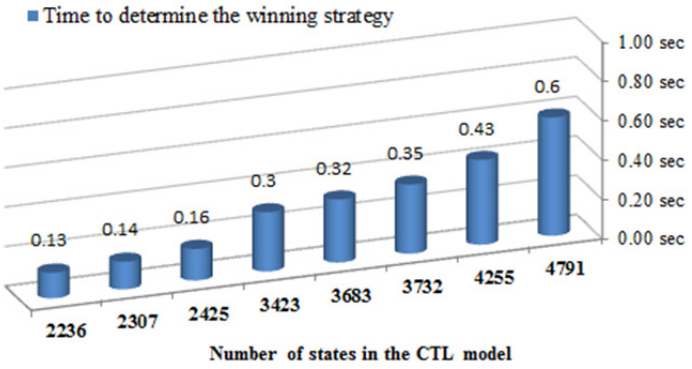

Figure 5.1.4: Evaluating the performance of the CTL model checker

6. An ATL model checking tool based on the algebraic methodology

Alur et al. introduced Alternating-time Temporal Logic (ATL), a logic designed for specifying requirements of open systems [AHK02]. An open system interacts with its environment and its behaviour depends on the state of the system as well as the behaviour of the environment. ATL is also widely used to reason about strategies in multiplayer games. The semantics of ATL is formalized by defining games such that the satisfaction of an ATL formula corresponds to the existence of a winning strategy.

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

Alternating-time Temporal Logic is a branching-time temporal logic that naturally describes computations of open systems, modelled by concurrent game structures.

6.1. The concurrent game structure

A *concurrent game structure* is defined 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 atomic *propositions* Γ; the ***labeling*** 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* in state q. For each player $a \in \Lambda$ and each state $q \in Q$, the *alternative moves* function d: $\Lambda \times Q \rightarrow 2^M$ relates the pair (a,q) with 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. The ***move function*** $D : Q \rightarrow 2^{\overline{M}}$, with $\overline{M}$ the set of all move vectors, is defined such that $D(q) \subseteq d_1(q) \times \ldots \times d_k(q)$ represents the set of move vectors at q. We denote by

$$D_a = \bigcup_{q \in Q} d_a(q) \tag{2}$$

the set of available moves of agent a within the concurrent 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 q' that results from state q if each player $a \in \Lambda$ choose the move j_a. The state q' is a successor of state q.

A *computation* of S is an infinite sequence $\lambda = q_0, q_1, \ldots$ such that q_{i+1} is a successor of q_i, $\forall i \geq 0$ [AHK02]. 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 [AHK02].

ATL syntax

We *denote* by $\mathcal{F}_S(\mathcal{A})$ the set of all well-formed ATL formulae 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 recursively 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 $\{\langle\langle\mathcal{A}\rangle\rangle \circ \varphi, \langle\langle\mathcal{A}\rangle\rangle \Box \varphi, \langle\langle\mathcal{A}\rangle\rangle \varphi_1 \, U \, \varphi_2\} \subseteq \mathcal{F}_S(\mathcal{A})$.

In the ATL logic the path quantifiers are parameterized by sets of players from Λ. The operator $\langle\langle\ \rangle\rangle$ is a path quantifier, and $\circ$ ('next'), $\Box$ ('always'), $\Diamond$ ('future') 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 [JB11]. 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* $U \varphi$.

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 [AHK02]:

- $j_a=f_a(\lambda[0,i])$ 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 $q \vDash \varphi$ 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 $\vDash$ is defined inductively as follows:

- for $p \in \Gamma$, $q \vDash p \Leftrightarrow p \in \gamma(q)$
- $q \vDash \neg\varphi \Leftrightarrow q \nvDash \varphi$
- $q \vDash \varphi_1 \vee \varphi_2 \Leftrightarrow q \vDash \varphi_1$ or $q \vDash \varphi_2$
- $q \vDash \langle\langle\mathcal{A}\rangle\rangle \circ \varphi \Leftrightarrow$ there exists a set $F_{\mathcal{A}}$ of strategies, such that for all computations $\lambda \in out(q, F_{\mathcal{A}})$, we have
  $\lambda[1] \vDash \varphi$ (the formula φ is satisfied in the successor of q within computation λ).
- $q \vDash \langle\langle\mathcal{A}\rangle\rangle \Box \varphi \Leftrightarrow$ 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 $\lambda[i] \vDash \varphi$ (the formula φ is satisfied in all states of computation λ).
- $q \vDash \langle\langle\mathcal{A}\rangle\rangle \varphi_1 \, U \, \varphi_2 \Leftrightarrow$ 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 $\lambda[i] \vDash \varphi_2$ and for all positions $0 \leq j < i$, we have $\lambda[j] \vDash \varphi_1$.

6.2. Implementation of an ATL model checker in ANTLR

In this section is presented an original approach represented by the generation of an algebraic compiler using ANTLR (Another Tool for Language Recognition) from our specification grammar of ATL.

The model checking problem for ATL is to determine whether a given system with its structure described by a concurrent game structure satisfies a given ATL formula. The purpose of our work is to implement a tool that allows to automatically checking for global system correctness.

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.

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

For implementation of the ATL compiler we choose the ANTLR (Another Tool for Language Recognition). ANTLR provides a framework for the generation of recognizers, compilers, and translators from grammatical descriptions [Parr07]. Using ANTLR as a generative tool had a major, positive impact on our overall productivity in development of the new ATL checker.

ANTLR supports infinite lookahead for selecting the rule alternative that matches the portion of the input stream being evaluated. The technical way of accomplishing this is that ANTLR supports LL(*) [Parr07], a feature which significantly enhanced parsing strength.

ANTLR takes as its input our ATL grammar - a precise description of the ATL language augmented with semantic actions - and generates source code files which are further extended and published through a Web service as server part of the ATL model checker tool.

A semantic action of a grammatical description from ATL grammar represents an action code written in Java – the target language of our ATL compiler. The action code is included inside the {} brackets, embedded into the generated parser and executed when an appropriate match in parsed input is made.

Also, ANTLR builds the Abstract Syntax Tree (AST), an intermediate tree representation of the parsed ATL input formula, which is simpler to process than the stream of tokens and can be efficiently processed multiple times.

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 $Q'=\{q\in Q \mid q \vDash \varphi\}$ – the set of states where the formula φ is satisfied. Translation of a formula φ of an ATL model to the set of nodes Q' over which formula φ is satisfied is accomplished by code included in semantic actions attached to production rules within specification grammar of ATL language. 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 ATL compiler $\mathcal{C}$, the attached actions define the core of the ATL model checker, i.e., the implementation of the ATL operators.

The ATL compiler $\mathcal{C}$ implements the following ATL model checking algorithm [Jam09]:

Algorithm 1. ATL model checking algorithm

Input: the concurrent game structure *S* and the formula φ

Output: $Q'=\{q\in Q \mid q \vDash \varphi\}$ – the set of states where the formula φ is satisfied.

function EvalA(φ) as set of states $\subseteq Q$

case φ=*p*:
  return $[p] = \{q \in Q \mid p \in \gamma(q)\}$;
case φ= ¬θ:
  return *Q*\EvalA(θ);
case φ=θ1∨θ2:
  return EvalA(θ1) ∪ EvalA(θ2);
case φ=θ1∧θ2:
  return EvalA(θ1) ∩ EvalA(θ2));
case φ=θ1→θ2:
  return (*Q*\EvalA(θ1)) ∪ EvalA(θ2);

case φ =⟨⟨𝒜⟩⟩○θ:
  return Pre(𝒜,EvalA(θ));
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;
case φ = ⟨⟨𝒜⟩⟩ θ1 *U* θ2:
  ρ:= ∅; τ:= EvalA(θ2); τ0:= EvalA(θ1);
  while τ ⊈ ρ do
      ρ := ρ ∪ τ;
   τ:=Pre(𝒜, ρ) ∩ τ0;
  wend
  return ρ;

The corresponding actions included in the ANTLR grammar of ATL language for implementing the ATL operators □, ◊, U and respectively ○ are presented in the table 6.2.1:

Implementation of the "□" operator	Implementation of the "◊" operator

'<<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;
}
	'<<A>>~' f=formula
	{
	HashSet Q = new HashSet(all_setS);
	HashSet r = new HashSet();
	HashSet p = $f.set;
	while (!r.containsAll(p))
	{
	r.addAll(p);
	p = Pre(r);
	p.retainAll(Q);
	}
	$set = r;
	}

Implementation of the "U" operator	**Implementation of the "○" operator**

'<<A>> ' a1= formula 'U' a2= formula
{
HashSet r = new HashSet();
HashSet p = $a2.set;
while (!r.containsAll(p))
{
r.addAll(p);
p = Pre(r);
p.retainAll($a1.set);
}
$set = r;
}
	'<<A>>@' f=formula
	{
	HashSet rez = Pre($f.set);
	$set = rez;
	}

Table 6.2.1 Semantic actions attached to production rules of ATL language grammar

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

The formula represents a term from a production rule of the ATL grammar and *p*, *r*, *a1*, *a2* are variables used in the internal implementation of the ATL compiler.

For a set $\mathcal{A}$ of agents, the implementation of most ATL operators implies the computation of function *Pre*($\mathcal{A}$, Θ), where $\Theta \subseteq Q$. The value returned by *Pre*($\mathcal{A}$, Θ) represents the set of states from which agents $\mathcal{A}$ can enforce the system into some state in Θ in one move.

In section 7 we made a implementation of the function *Pre*() using SQL statements, ready to be executed on a high-speed database server.

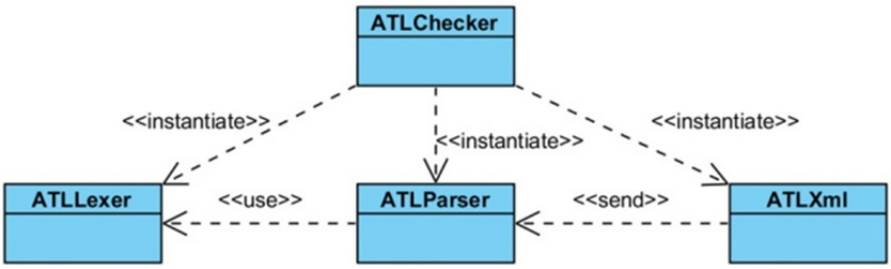

Figure 6.2.1: Class diagram of the ATL Checker.

In figure 6.2.1 is presented the class diagram of the ATL model checker implementation. Classes ATLParser and ATLLexer are generated by ANTLR using as input our grammar of the ATL language. The role of class ATXml is to decode the XML representation of the ATL model in order to send it to the parser, along with the ATL formula which must be evaluated. The ATLChecker class contains the Web service operations which are invoked by the client, having as parameters an ATL model, an ATL formula and a set $\mathcal{A}$ of agents.

6.3. Using Relational Algebra in model checking algorithm

For a concurrent game structure *S* presented in section 6.1, can be defined a directed multi-graph $G_S = (X,U)$, where $X=Q$, and $(b,e) \in U \Leftrightarrow \exists \langle j_1,\ldots,j_k\rangle \in D(b)$ 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, $(B{:}b, M_1{:}j_1, \ldots, M_k{:}j_k, \text{E:e}) \in R_S \Leftrightarrow \langle j_1,\ldots,j_k\rangle = L((b,e))$.

For a set $\mathcal{A}$ of m agents, $\mathcal{A} \subseteq \Lambda$, $\mathcal{A} = \{i_1, \ldots, i_m\}$, we define $R_S(\mathcal{A}) = \pi_{B,M_{i_1},\ldots,M_{i_m},E}(R_S)$ where $i_l \in \mathcal{A}, l \in \{1,\ldots,m\}$ 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: $i \circ j = i \,||\, ',' \,||\, j$.

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 $\overline{\Theta}$.

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

Algorithm 2. Computing $Pre(A,\Theta)$ 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

$$\rho_X(R_L^{\Theta}(\mathcal{A})) \underset{x.B = y.B \wedge x.LABEL = y.LABEL}{\bowtie} \rho_Y(R_L^{\overline{\Theta}}(\mathcal{A})) = R_L^{\Theta,\overline{\Theta}}(\mathcal{A})$$

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 $Pre(\mathcal{A},\Theta)$ 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
```

6.4. An ATL model ATL for the critical section problem solved using a mutex

In [Rus02] is presented a CTL model for two processes competing for entrance into a critical section.

In the following, we present an original 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 6.4.1 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:

- I_i – the process i is in ***Idle*** state, $i = \overline{1,2}$;
- W_i – the process i is in ***Waiting*** state (it is waiting to enter in critical section), $i = \overline{1,2}$;
- E_i – the process i is in ***Executing*** state (it is executing the code from critical section), $i = \overline{1,2}$;
- L_i – the mutex is owned (***Locked***) by the process i, $i = \overline{1,2}$;
- F – the mutex is not owned by any process (it has ***Freed***).

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 (lock the mutex);*
- *e – a request to execute code from the critical section;*
- *i – there is no a request (idle);*

- *f – release (free) the mutex, leave the critical section;*
- *pd – permission for agent 1, deny for agent 2;*
- *dp – permission for agent 2, deny for agent 1;*
- *p- – permission for agent 1, the agent 2 is idle (no request);*
- *-p – permission for agent 2, the agent 1 is idle (no request).*

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

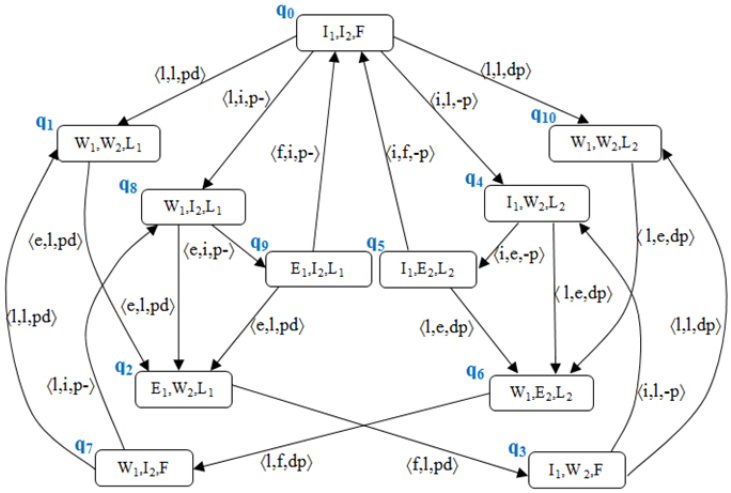

Fig. 6.4.1. ATL model for two processes competing for entrance into a critical section.

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 6.4.1. 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 6.4.1.

Example 6.4.1.

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

$$W_1 \Rightarrow not\ (\langle\langle \mathcal{A} \rangle\rangle \#\ (not\ E_1)) \quad (3)$$

with its signification described in Table 6.4.1. The model checking algorithm will require some calls of function *Pre()* with certain arguments. In Table 6.4.2. are presented two computations of function *Pre()*:

	$\Theta = \{0,3,4,5,6,7,10\}$	
	$\mathcal{A}=\{1\}$	$\mathcal{A}=\{2\}$
$\pi_{x.B,y.LABEL} R_L^{\Theta,\bar{\Theta}}(\mathcal{A})$	**B** / **LABEL** 0 / NULL 0 / 1 2 / NULL 3 / NULL 4 / NULL 5 / NULL 6 / NULL 9 / NULL 10 / NULL	**B** / **LABEL** 0 / 1 2 / NULL 3 / NULL 4 / NULL 5 / NULL 6 / NULL 9 / NULL 10 / NULL
$Pre(\mathcal{A},\Theta)$	$\{0,2,3,4,5,6,9,10\}$	$\{2,3,4,5,6,9,10\}$

Table 6.4.2. Computations of function $Pre(\mathcal{A},\Theta)$ when Checking the ATL Formula (3)

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

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 $0 \notin Pre(\mathcal{A}, \Theta)$.

Example 6.4.2

For the same ATL model, described in figure 6.4.1, we consider the following formula:

$$not\ (\langle\langle \mathcal{A} \rangle\rangle \sim\ (not\ (I_1 \Rightarrow \langle\langle \mathcal{A} \rangle\rangle @\ W_1))) \quad (4)$$

with its signification also described in Table 6.4.1. In table 6.4.3. are presented computations of function *Pre()* needed for checking the ATL formula (4):

	$\Theta = \{1,6,7,8,10\}$	
	$\mathcal{A}=\{1\}$	$\mathcal{A}=\{2\}$
$\pi_{x.B,y.LABEL} R_L^{\Theta,\bar{\Theta}}(\mathcal{A})$	**B** / **LABEL** 0 / NULL 3 / NULL 4 / NULL 5 / NULL 6 / NULL 7 / NULL 10 / NULL	**B** / **LABEL** 0 / 1 0 / NULL 3 / 1 4 / e 5 / NULL 6 / NULL 7 / NULL 10 / NULL
$Pre(\mathcal{A},\Theta)$	$\{0,3,4,5,6,7,10\}$	$\{0,5,6,7,10\}$

Table 6.4.3. Computations of function $Pre(\mathcal{A},\Theta)$ when Checking the ATL Formula (4)

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 $4 \notin Pre(\mathcal{A}, \Theta)$.

7. The architecture of the ATL model checker tool. Applications. Performance evaluation

We choose to use Web Services technology in our implementation of the ATL model checker in order to make the core of out tool, the ATL compiler, accessible to various clients.

A Web service represents a standardized way for an application to expose its functionality over the Web or communicate with other applications over a network, regardless of the platform or operating system upon which the clients of the service are implemented. Thus, our Java implementation of the ATL Checker can be invoked easily through a Web service by a C# client who provides an intuitive graphical interface for interactive design of ATL models. We called the client application ATL Designer.

Other reason for the deployment of ATL Checker on server side is represented by the internal implementation of the *Pre()* function, described in section 7, which are using for its computation SQL queries. Our Web service is using GlassFish as a Web container, and MySQL or SQL Server as database servers.

For a better understanding of the ATL model checking process, in figure 7.1 is represented the Use Case Diagram of our model checker:

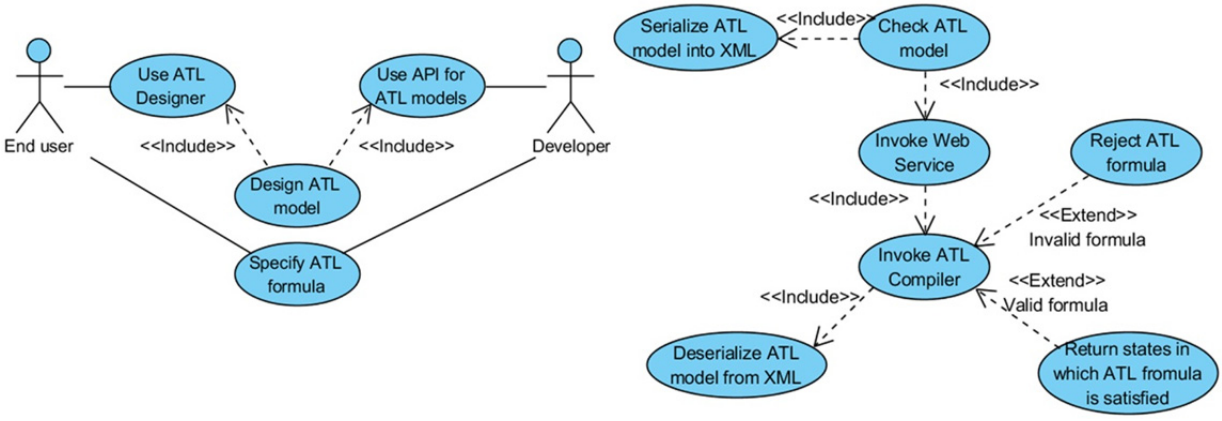

Figure 7.1: The Use Case Diagram of the ATL model checker

The Web service will receive from a client the XML representation of an ATL model and an ATL formula. After deserialization, the original form of the ATL model is passed to the ATL compiler generated by ANTLR using our ATL extended grammar. For a syntactically correct formula, the compiler will return as result the set of states in which the formula is satisfied. If the ATL formula is not valid, the Web service will return a message describing the error.

In order to notify the client about possible syntactical errors found in the verified ATL formula, we must override the default behaviour of the ANTLR error-handling. We install our error-handling in lexer and parser:

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

Finally, we instruct ANTLR to throw the error, allowing the Web service to send it to the client:

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

ATL Designer, the client part of our tool, allows an interactive construction of the concurrent game structures as directed multi-graphs. For internal representation of an ATL model as a directed multi-graph, our implementation is based on data structures provided by [Ebe87]. Thus, the 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 a dynamically graphical development of the ATL models as we can see from the figure 7.2:

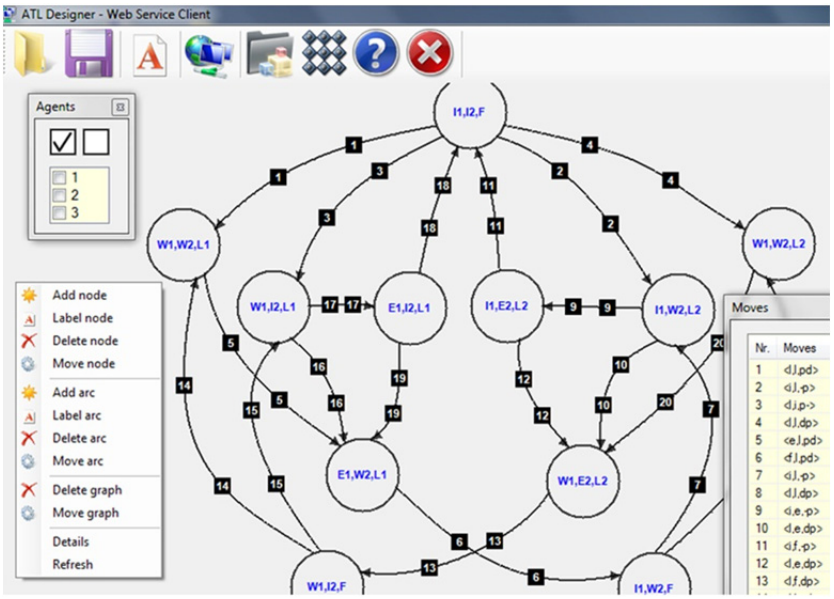

Figure 7.2: Building an ATL model in ATL Designer

In figure 7.2, the numbered labels of edges are associated with move vectors of agents, and can be assigned in the "Moves" window of the ATL Designer.

In addition, the ATL Designer interface allows several configurations:

Icon	Signification
	Allows selection of the Web server: local or remote (Internet). Service-location details for web-service access can be found at: http://use-it.ro
	The status bar button allows selection of the implicit Internet server. The toolbar button allows setting of the database connection string.
	In case of huge models, is recommended to activate their compression before sending them to the Web Service.

Table 7.1: ATL Designer configuration from its interface

For testing purposes, the ATL model checker described in this paper is available online via two Web services hosted by *use-it.ro* and respectively by *mcheck-useit.rhcloud.com*.

7.1. System architecture of the ATL model checker tool

In order to provide an overview of the system architecture of the ATL checker tool presented in this paper, we chose a UML package diagram, presented in figure 7.1.1:

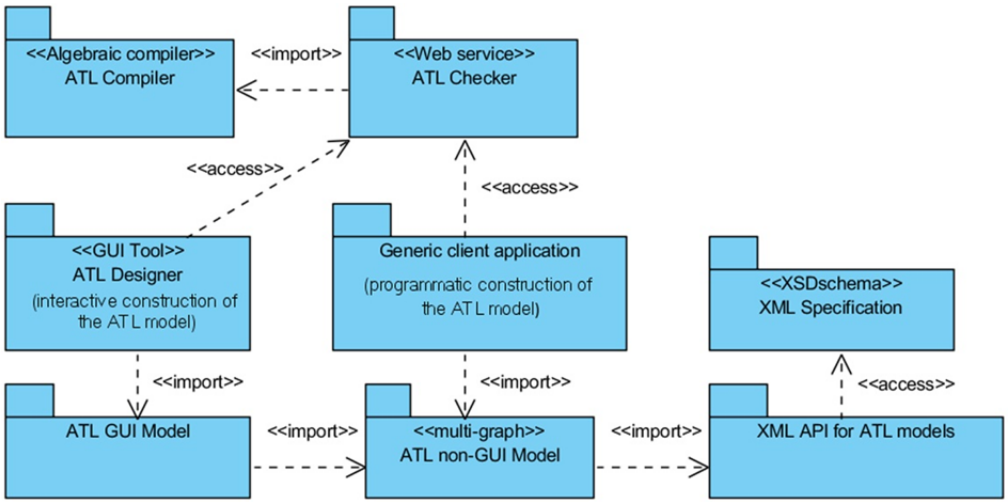

Figure 7.1.1 System architecture of the ATL model checker tool

The ATL model checker tool contains the following packages:

- The ATL Compiler embedded into the Web Service (ATL Checker);
- The GUI client application used for interactive construction of the ATL models as directed multi-graphs (ATL Designer);
- In case of huge ATL models, with many states, is required the use a programmatic construction of these models. The *ATL non-GUI model* package contains classes used for internal representation of an ATL model as a directed multi-graph based on symmetrically stored forward and backward adjacency lists;
- The *XML API for ATL models* package contains classes needed to encode the ATL model into XML. The main part of its code was generated using Microsoft Xml Schemas/Data Types support utility (xsd.exe) having as input 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 (drawing arcs by Bézier curves, etc.).

7.2. Designing a game strategy using model checking

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 [HW02].

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

As an example we consider the Tic-Tac-Toe (called TTT for short in the rest of this paper) game. 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 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 $b\in\Lambda\backslash\{a\}$. 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.

Modelling the Game

We transform the original problem into an ATL model checking problem. More specifically, we want to determine a strategy $f_a : Q^+ \to D_a$ 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 7.2.1:

0	1	2
3	4	5
6	7	8

Fig. 7.2.1: 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 $\overline{T} = 1$ signify 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 state $q \in Q$, if there are possible moves, is defined as

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

Analogous are defined the possible moves of 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:

$$\overline{T} \in \gamma(q) \Rightarrow \overline{3-T} \in \gamma(q')$$

where $T \in \{1,2\}$.

Algorithm to determine the optimal strategy

Assuming that the game is in the state q ∈ Q, we denote by succ(q) ⊆ Q the set of states immediately following the state q in the tree modelling the concurrent game structure.

The strategy of player 1 can be expressed by the following algorithm:

Step 1	Determines all states from the model satisfying the formula: ⟪1⟫ ◊ (111), to choose the move which favours wining of the game in the future. We denote this set with *WIN1*.
Step 2	Determines all states from the model satisfying the formula: ⟪2⟫ ○ (222), to prevent player 2 to win on the next move. We denote this set with *WIN2*.

Step 3	If $(WIN1 \setminus WIN2) \cap succ(q) \neq \emptyset$ then
	Choose randomly a state q from the resulting set.
	Else If $succ(q) \setminus WIN2 \neq \emptyset$ then
	Choose randomly a state q from the resulting set.
	Else
	Choose randomly a state q from the set $succ(q)$.
	End If
	Set q as current state.

Step 4	If $\overline{111} \in \gamma(q)$ then STOP. The player 1 has won. If the board is full is declared equality and STOP, else after the move of player 2 go to step 1 (if player 2 has not won or the board is not full).

In the following we present a game scenario implemented using the ATL model checker API.

At first move, the computer (player 1) chooses the position 0. After the player 2 moves, is constructed the ATL model of the game. This model has 4791 states and 4790 transitions.

In figure 7.2.2 can be seen that player 1 has determined the winning strategies, having three alternatives to win the game, from which is chosen randomly one.

Figure 7.2.2: The move of player 1 (in position 4) which follows a winning strategy

Finally, can be seen that player 2 could not avoid defeat, the player 1 choosing the only option left to win:

Figure 7.2.3: The player 1 (computer) won

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 analyze 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:

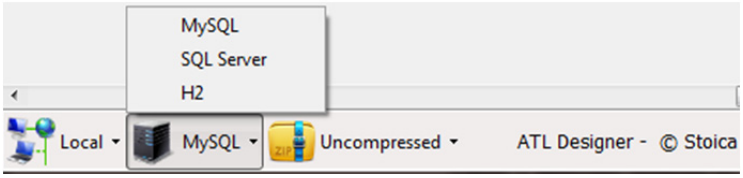

Figure 7.2.4: The selection of the database server

We have found important performance penalty due to the clause IN from the query presented in algorithm 3 from section 6, 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 7.2.1 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 [H2DB] 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_DELA
Y=60
```

In table 7.2.2 and respectively in the figure 7.2.5 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 7.2.2 A comparative analysis of impact of database servers in performance of ATL model checker

In [OM03] 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 [AHK02].

Results from [OM03] 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 7.2.2, 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.).

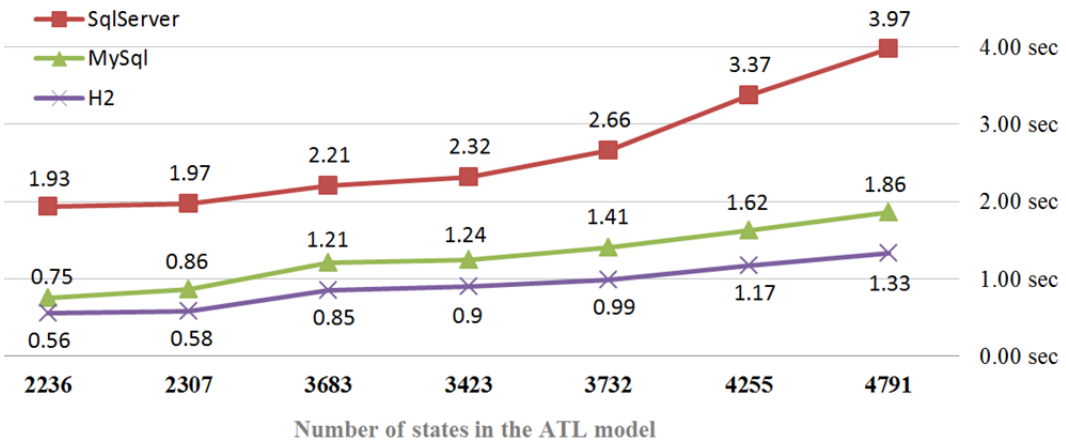

Figure 7.2.5: The performance of ATL model checker related to database server used

In [Rua08] 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. [AHMQRT98]. 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.

7.3. Verification of JADE agents using ATL model checking

In the end of the chapter, using components of our tool, we showed how ATL model checking technology can be used for automated verification of multi-agent systems, developed with JADE.

One of the main drawbacks of employing ATL logic in the automated verification of multi-agent systems using previous approaches consists in necessity of translate the programs written in specific modelling languages to the programming language used in the real implementation.

Our approach eliminates this problem by allowing a transparent building of the ATL model at runtime, using the native language of JADE agents (Java).

We build an ATL model suited for FSM (Finite State Machine) - driven behaviour of a JADE agent. This model will help us to elaborate the mapping rules between ATL and JADE concepts. ATL Library will be used to validate the design of JADE agents having FSM-behaviours, in other words, to see that no incorrect scenarios arise as a consequence of a bad design.

7.3.1. A formal model of the FSMBehaviour

In the following we present a model for FSM-driven behaviour of a JADE agent, implemented by FSMBehaviour class. This model will help us to elaborate the mapping rules between ATL and JADE concepts.

A JADE finite state machine is a tuple FSM=(QFSM, Π, π, q0, F, t, δFSM) where:

- Q_{FSM} is a finite, non-empty set of *states*;
- denotes the finite set of *state names*;
- π. $Q_{FSM} \rightarrow \Pi$ is called the ***labelling*** function, defined as follows: for each state $q \in Q_{FSM}$, $\pi(q) \in \Pi$ is the *name* of state q;
- q_0 is an element of Q_{FSM}, the initial state;
- $F \subseteq Q_{FSM}$ is the set of final states;

- t: $Q_{FSM} \rightarrow 2^{\mathbb{Z} \cup \{default\}}$ is called the ***terminating*** function, where for each state $q \in Q_{FSM}$, $t(q) \subseteq \mathbb{Z} \cup \{default\}$ represents the set of admissible termination codes of the state q;

The transition function $\delta_{FSM}(q, j)$, associates to each state $q \in Q_{FSM}$ and each termination code j of q the state that results from state q if the child behaviour associated with the state q returns at finish the value j.

The behaviour of an FSM is more easily understood when this is represented graphically in the form of a state transition diagram. The control states are represented by circles, and the transition rules are specified as directed edges. Each transition from a state q is labelled by termination code of q that triggers the transition. The arc without a source state denotes then initial state of the system (state q_0).

During one reaction of the FSM, one transition is triggered, chosen from the set of admissible transitions (outgoing transitions from the current state), so that label of transition matches the terminating code of the current state. The FSM goes to the destination state of the triggered transition.

If terminating code of the current state $q \notin F$ is not explicit associated with an admissible transition, then:

- if exist the admissible transition labelled with *default*, this transition (called *implicit transition*) will be triggered;
- else FSM goes in an inconsistent state.

In case if FSM arrive in a state q ∈ F, after completeness of activities from that state, execution of finite state machine is stopped.

7.3.2. ATL model of the FSMBehaviour

For a JADE finite state machine defined in section 7.3.1, the equivalent concurrent game structure $S=\langle \Lambda, Q, \Gamma, \gamma, M, d, \delta \rangle$ is defined as follows:

- There is only one agent, i.e. $\Lambda = \{1\}$;
- The set of states is $Q = Q_{FSM}$;
- The finite set of *propositions* is defined by $\Gamma = \Pi \cup \{ *FINAL* \}$;
- The labelling function γ: $Q \rightarrow 2^{\Gamma}$ is defined as follows:

$$\gamma(q) = \begin{cases} \pi(q) \; for \; q \in Q \setminus F \\ \pi(q) \cup \{*FINAL*\} \; for \; q \in F \end{cases}$$

- The nonempty finite set of moves M contains all admissible termination codes, i.e.:

$$M = \bigcup_{q \in Q} t(q)$$

- the ***alternative moves*** function $d: \Lambda \times Q \rightarrow 2^M$ is defined by $d(1, q) = t(q) \forall q \in Q$
- the transition function δ is defines as follows:

$$\delta(q, < j >) = \delta_{FSM}(q, j) \forall q \in Q \; and \; \forall j \in t(q)$$

7.3.3. Using ATL for verification of the FSM - driven behaviour of a JADE agent

For a given JADE FSMBehaviour, the ATL model checking is done in two steps:

1. For the beginning, the corresponding ATL is constructed following rules described in section 7.3.2
2. Then, a given specification (ATL formula) representing a desired behavioural property is verified to hold for the model obtained at step 1.

Using ATL Library [CS13] to perform ATL model checking, we can detect error states (the states of the model where the ATL formula does not hold) and then we can correct the given model or design.

8. Conclusions and future directions

As future research we intend:

- Creating libraries which include several models CTL / ATL that can be verified and studied by user;
- Creating a CTL model checker extension which include the time restrictions;
- Developing a symbolic CTL model checker;
- Creating an ATL model checker extension which include the time restrictions;
- Developing a symbolic ATL model checker;
- Although the model checking tools developed so far were used for the verification of complex systems, a major limitation of this approach is that those tools can only verify the correctness of the system specifications. In other words, if errors are identified by a model checker tool within a specified system, the task of system correction is altogether left to the system designers. Accordingly, model checking is generally only used to verify whether a system properly holds but without change of the system if the verification fails. Automatic correction of the system was addressed in some papers [SW96, DNM06, CPPB08], for some specific cases. A possible research direction is the automatic modification of a model when its verification failed.
- The development of theoretical models and extensions of temporal logics discussed within the thesis to extend the scope of the application of the model checking technology to the software systems of great actuality: applications/Web Services, semantic web services (Semantic Web) [Mar04], distributed/autonomous database management systems for web applications, etc.

As a future direction of our research, we propose an extension of the model checking verification tools presented in this thesis to generate automatic (or at least assisted) the specifications to be verified (formulas currently expressed in the language associated with temporal logic), through a friendly and interactive graphical interface.

Each of these issues requires complex challenges, both theoretically and practically, but the importance of the research field approached within the thesis represents a sufficient motivation to address them continuing results achieved by now.

We manifest the hope that the original theoretical results obtained in this thesis, integrated and exploited in the implementations of our model checking tools, can contribute to a long expected desiderate: integration of the formal verification techniques in the current cycle of design and development of software systems.

REFERENCES

[ADKKM05] N. Amla, X. Du, A. Kuehlmann, R.P. Kurshan and K.L. McMillan. An Analysis of SAT-based Model Checking Techniques in an Industrial Environment. In *Charme*. pages 254–268, 2005. http://www.kenmcmil.com/pubs/CHARME05.pdf

[AH06] J.A. Anderson and T. J. Head. Automata theory with modern applications. *Cambridge University Press*, pages 105–108, 2006. ISBN 9780521848879.

[AH96] R. Alur and T.A. Henzinger. Reactive modules. *In Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pages 207–218, 1996.

[AHK02] R. Alur, T.A. Henzinger and O. Kupferman. Alternating -Time Temporal Logic. *Journal of the ACM*. Vol. 49, No. 5, pages 672–713, 2002.

[AHMQRT98] R. Alur, T.A. Henzinger, F.Y.C Mang, S. Qadeer, S.K. Rajamani and S. Tasiran. Mocha: Modularity in Model Checking. *Proceedings of the Tenth International Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science 1427, Springer, pages 521-525, 1998.

[ANTLR] http://www.antlr.org/testimonials.html

[BBFLPPS01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci and Ph. Schnoebelen. Systems And Software Verification Model-Checking Techniques And Tools. *Springer-Verlag And Heidelberg Gmbh & Co. Kg*, Berlin, 2001. ISBN 3-540-41523-8.

[BBCR10] J. Barnat, L. Brim, M. Češka and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). *In Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC)*, pages 4–7, IEEE, 2010.

[BBR10] J. Barnat, L. Brim and P. Ročkai. Scalable Shared Memory LTL Model Checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2): pages 139–153, 2010.

[BBPESR10] Brad Bingham, Jesse Bingham, Flavio M. de Paula, John Erickson, Gaurav Singh and Mark 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, pages 28–36, 2010.

[BCCFZ99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 317–320, 1999.

[BCM92] J.R. Burch, E.M. Clarke and K.L. McMillan. Symbolic Model Checking 10^{20} States and Beyond. *Information and Computation*, Academic Press Inc., New York and London, Vol. 98, pages 142–170, 1992.

[BCTR13] F. Bellifemine, G. Caire, T. Trucco, G. Rimassa, *JADE programmer's guide*, http://jade.tilab.com, 2013

[BFBD02] F. M. Boian, C. M. Ferdean, R. F. Boian, R. C. Dragos. Programare concurenta pe platforme Unix, Windows, Java. *Editura Albastra - grupul Microinformatica*, ISBN 973-650-072-1, pages 420, Cluj, 2002.

[Bison] http://dinosaur.compilertools.net/bison/

[BK08] Christel Baier, Joost-Pieter Katoen, *Principles of Model Checking*, The MIT Press, Cambridge, Massachusetts, London, England, 2008

[BMP90] C.J. Bergman, R.D. Maddux and D.L. Pigozzi. Algebraic Logic and Universal

Algebra in Computer Science. *Lecture Notes in Computer Science*, Springer, LNCS, Vol. 425, pages 209–225, 1990.

[Boi11] F.M. Boian. Servicii Web; Modele; Platforme;Aplicaţii. *Seria 245 PC. Editura Albastră*, Cluj-Napoca, pages 1–382, 2011.

[Bov] Jean Bovet. ANTLRWorks: The ANTLR GUI Development Environment, http://www.antlr.org/works/index.html

[BP07] Jean Bovet and Terence Parr. An ANTLR Grammar Development Environment, http://www.antlr.org/papers/antlrworks-draft.pdf

[BP08] M. P. Bhave and S. A. Patekar. Programming With Java. *Pearson Education India*, pages 1–748, 2008. ISBN 8131720802, 9788131720806.

[BS03] M. Boyer and M. Sighireanu. Synthesis and verification of constraints in the PGM protocol. *FME 2003: Formal Methods. Proceedings of International Symposiumof Formal Methods Europe.* Springer-Verlag, Pisa (Italy), September, pages 264–281, 2003.

[BS84] R.A. Bull and K. Segerberg. Basic Modal Logic. *Handbook of Philosophical Logic*. Vol. 2. Kluwer, pages 1–88, 1984.

[BZ02] C. Baral, and Y. Zhang. The complexity of Model Checking for Knowledge Update, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.2843, 2002.

[BZ05] C. Baral and Y. Zhang. Knowledge updates: semantics and complexity issues. *Artificial Intelligence*. Vol. 164, Issues 1–2 May 2005, pages 209–243, 2005.

[CPPB08] Laura F. Cacovean, Iulian Pah, Emil M. Popa and Cristina I. Brumar. Algorithm and an elevator control system example for the CTL model checker. *ICE-B 2008, International Conference on E-Business, Porto, Portugal, July 26-29*, pages 77–80, 2008, ISBN: 978-989-8111-58-6

[Ca09] Laura Florentina Cacovean. An Algebraic Specification for CTL with Time Constraints. *First International Conference on "Modelling and Development of Intelligent Systems"*, MDIS'09, Sibiu, Romania, pages 46–55, 2009. ISSN 2067 - 3965.

[CC94] S.V. Campos and E.M. Clarke. Real-time symbolic model checking for discrete time. Models. *Theories and Experiences for Real-Time System Development*. World Scientific, pages 129–145, 1994.

[CCGPRST02] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Rovere, R. Sebastiani and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. *Computer Science Department*. Technical Report Paper 430, pages 1–5, 2002. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.8023

[CCGR02] A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri. NuSMV: a new symbolic model checker. *STTT International Journal on Software Tools for Technology Transfer*. Springer Verlag, pages 410–425, 2002.

[CCGR99] A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri. NuSMV: A new symbolic model verifier. *Proceedings of the 11th International Conference on Computer Aided Verification.* CAV '99, pages 495–499, 1999. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.2411

[CCKSVW02] P. Chauhan, E. Clarke, J. Kukula S. Sapra, H. Veith and D. Wang. Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis. *Proceeding FMCAD '02 Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*. Springer-Verlag London, pages 33–51, 2002. ISBN:3-540-00116-6.

[CDEG03] Marsha Chechik, Benet Devereux, Steve Easterbrook and Arie Gurfinkel. Multi-Valued Symbolic Model-Checking. *Journal of ACM Transactions on Software Engineering and Methodology*, Vol. 12, Issue 4, pages 371 – 408, 2003.

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.7144

[CE82] E.M. Clarke and E.A. Emerson. Design and Synthesis of synchronization skeletons for branching time temporal logic. *In Logic of Programs*, 1981. Lecture Notes in Computer Science, No. 131, Springer-Verlag, 1982.

[CES86] E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*. Vol. 8 (2), pages 244–263, 1986.

[CGL96] E. Clarke, O. Grumberg and D. Long. Model checking. *Proceedings of the International Summer School on Deductive Program Design* Marktobedorf, Germany, 1994. In *M. Broy, "Deductive Program Design"*, NATO ASI, Series F, vol. 152, Springer Verlag, 1996.

[CGP00] E. Clarke, O. Grumberg and D.A. Peled. Model Checking. *MIT Press*. Cambridge, 2000, pages 1–325.

[CL07] E. Clarke, F. Lerda. Model Checking: Software and Beyond, *Journal of Universal Computer Science*, vol. 13, no. 5, pp. 639-649, 2007

[CPBP08] L.F. Cacovean, E.M. Popa, C.I. Brumar and I. Pah. An application CTL formula based on Problem Solving Methodology. *New Aspects of Computers from Proceedings of the 12th WSEAS International Conference of Computers*. Heraklion, Greece, pages 218–223, 2008. ISSN: 1790-5109, ISBN: 978-960-6766-85-5.

[CS08] Laura F. Cacovean and Florin Stoica. Algebraic Specification Implementation for CTL Model Checker Using ANTLR Tools. *WSEAS International Conferences, Computers and Simulation in Modern Science - Volume II*, Bucharest, Romania, pages 45–50, 2008. ISSN: 1790-5117, ISBN: 978-960-474-032-1

[CS09] Laura F. Cacovean and Florin Stoica. CTL Model Update Implementation Using ANTLR Tools. *Proceedings of the 13th WSEAS International Conference on COMPUTERS*, Rhodos, Greece, pages 169–174, 2009. ISSN: 1790-5109, ISBN: 978-960-474-099-4

[CS10] Laura Florentina Cacovean and Florin Stoica. Modeling the Broker Behavior Using a BDI Agent. *Proceedings of the 14th WSEAS International Conference on Computers (CSCC)*. Corfu, Grecia, pages 699–703, 2010.

[CS13] L. F. Cacovean, F. Stoica, WebCheck – ATL/CTL model checker tool, http://use-it.ro

[CSS11] L.F. Cacovean, F. Stoica and D. Simian. A New Model Checking Tool. *Proceedings of the European Computing Conference (ECC '11)*. Paris, France, April 28-30, 2011, pages 358–364, 2011

[DEW04] K. Denecke, M. Erne and S.L. Wismath. Galois Connections and Applications. *Kluwer Academic Publishers*, Dordrecht, pages 1–501, 2004.

[DNM06] A.L. Dennis, P. Nogueira and R. Monroy. Proof-directed Debugging and Repair. Local Proceedings of the 7th Symposium on Trends in Functional Programming (TFP'06), Nottingham, UK, pages 131–140, 2006.

[Drools] http://blog.athico.com/2007/06/interview-with-antlr-30-author-terrence.html

[Ebe87] J. Ebert. A Versatile Data Structure for Edge-Oriented Graph Algorithms. *Communications of the ACM*, Vol. 30 No. 6, pages 513-519, 1987.

[EMSS91] E.A. Emerson, A.K. Mok, A.P. Sistla and J. Srinivasan. Quantitative temporal reasoning. *CAV '90 Proceedings of the 2nd International Workshop on Computer Aided Verification*. Springer-Verlag, London, pages 136–145, 1991.

[Eme90] E. A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, Volume B: Formal Models and Sematics. *Elsevier and MIT Press*, pages

995–1072 , 1990.

[EP02] Cindy Eisner and Doron Peled. Comparing Symbolic and Explicit Model Checking of a Software System. *Lecture Notes in Computer Science* 2318, pages 230–239, 2002.

[FFST11] Dieter Fensel, Federico Michele Facca, Elena Simperl and Ioan Toma. Semantic Web Services. *Springer-Verlag Berlin HeidelBerg*, pages 1–357, 2011.

[For02] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time. *In ICFP'02*, pages 36–47. ACM Press, 2002

[For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. *In POPL'04*, pages 111–122. ACM Press, 2004.

[Geo] G. Georgescu. Algebra Logicii - Logica Algebrică (I). http://egovbus.net/rdl/articole/No1Art34.pdf

[GrStr12] GraphStream. A Dynamic Graph Library, http://graphstream-project.org, 2012.

[GV04] P. Gammie and R. Van Der Meyden. MCK - Model Checking the Logic of Knowledge. *Computer Aided Verification.* Springer, Berlin, pages 479–483, 2004.

[H2DB] H2 Database Engine, http://www.h2database.com/html/performance.html.

[Hau99] R.R. Hausser. Foundations of Computational Linguistics. *Springer–Verlag*, Berlin, pages 1–534, 1999.

[Her10] M. Herschel. Database Systems I. Database Systems Group, University of Tübingen, http://db.inf.uni-tuebingen.de/teaching/ss10/db1/05_relational_algebra.pdf, 2010.

[HG98] Paul Halmos and Steven Givant. Logic as Algebra. *The Mathematical Association of America*. Vol. 21, pages 1-152, 1998.

[HH85] H. Herrlich and M. Husek. Galois connections. *Mathematical Foundation of Programming Semantics*. LNCS 239, *Springer-Verlag*, pages 122–134, 1958.

[Hol04] G.J. Holzmann. The SPIN Model Checker. *Primer and Reference Manual*. *Addison-Wesley*, pages I-XII, 1–596, 2004.

[Hol97] G. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering* 23, pages 279–295, 1997.

[HR00] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, pages 1–405, 2000.

[HR04a] Gerard J. Holzmann and Joshi Rajeev. Model-Driven Software Verification. SPIN 2004, pages 76–91, 2004.

[HR04b] M. Huth and Mark Ryan. *Logic in Computer Science* (Second Edition), Cambridge University Press, pages 1–440, 2004.

[Hu95] A.J. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*, PhD thesis, Stanford University, 1995.

[HW02] W. van der Hoek, M. Wooldridge, Tractable multiagent planning for epistemic goals, in *Proceedings of AAMAS-02*, pp. 1167-1174, ACM Press, 2002.

[JADE] Java Agent Development Framework (JADE), http://jade.tilab.com/

[Jam09] W. Jamroga. Easy Yet Hard: Model Checking Strategies of Agents. Computational Logic in Multi-Agent Systems. *Lecture Notes in Computer Science*, Vol. 5405, *Springer Berlin Heidelberg*, pages 1–12, 2009.

[JB11] W. Jamroga and N. Bulling. Comparing variants of strategic ability. *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume*

One, pages 252–257, 2011.

[KM08] Fred Kröger and Stephan Merz. Temporal Logic and State Systems. *Springer Verlag Berlin, First Edition*, pages 1–433, 2008.

[KP05] M. Kacprzak and W. Penczek. Fully symbolic Unbounded Model Checking for Alternating-time Temporal Logic. *Journal Autonomous Agents and Multi-Agent Systems archive*, Vol. 11, Issue 1, pages 69 – 89, 2005.

[Kri63] Saul Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16, pages 83–94, 1963.

[Kri93] Saul Kripke. Semantical analysis of modal logic I: Normal modal propositional calculi. *Zeitschrift f. Math. Logik und Grundlagen d. Math.,* 9, pages 67–97, 1993.

[KW02] Marcus Kracht and Frank Wolter. Advances in Modal Logic, Vol 3. *CSLI lecture notes Volumul 3 din Advances in Modal Logic, Advances in Modal Logic*. World Scientific, pages 1–424, 2002.

[Kle06] Kevin C Klement. Propositional Logic. In *James Fieser and Bradley Dowden (eds.), Internet Encyclopedia of Philosophy,* http://www.iep.utm.edu/prop-log, 2006.

[LR06] A. Lomuscio and F. Raimondi. Mcmas: A model checker for multi-agent systems. *In Proceedings of TACAS 06, volume 3920 of Lecture Notes in Computer Sciences*, pages 450–454. Springer-Verlag, 2006.

[LS99] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of SPIN 1999, volume 1680 of LNCS. Springer-Verlag*, pages 22–39, 1999.

[LSDLS12] Yi Li, Jing Sun, Jin Song Dong, Yang Liu and Jun Sun. Translating PDDL into CSP# - The PAT Approach, *ICECCS '12 Proceedings of the 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, pages 240-249, IEEE Computer Society Washington DC, 2012

[LST03] Flavio Lerda, Nishant Sinha and Michael Theobald. Symbolic Model Checking of Software, *Electronic Notes in Theoretical Computer Science*, Volume 89, Issue 3, pages 480–498, 2003.

[Lyo06] John Lyons. Natural Language and Universal Grammar: Volume 1: Essays in Linguistic Theory. *Cambridge University Press*, pages 1–308, 2006

[MA03] K. McMillan and N. Amla. Automatic abstraction without counter-examples. *Proceeding of TACAS'03 Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems. Springer-Verlag Berlin, Heidelberg*, pages 2–17, 2003.

[Mar04] Martin, D., et al. OWL-S: Semantic markup for web services. *W3C Member Submission*, November, 2004, http://www.w3.org/Submission/OWL-S/

[Mas03] Franceschet Massimo. staff.science.uva.nl/~schlobac/Teaching/AR2003/massimo_1.pdf, 2003

[MB79] S. Mac Lane and G. Birkhoff. Algebra. *MacMillan Publishing Co., Inc., second edition*, pages 1–586, 1979.

[MBT00] Ali Abbas Mir, Subhashini Balakrishnan and Sofine Tahar. Modeling and Verification of Embedded Systems using Cadence SMV. *Conference on Electrical and Computer Engineering, 2000 Canadian*, Vol. 1, pages 179–183, 2000.

[McG03] James McGovern. Java Web Services Architecture. *Elsevier Science, SUA,* pages 1-833 pagini, 2003

[McM93] K. L. McMillan. Symbolc Model Checking. *Kluwer Academic Publishers, Boston, Dordrecht*, Londra, pages 1–194, 1993.

[Mir00] Ali Abbas Mir. Subhashini Balakrishnan and Sofine Tahar. Modeling and Verification of Embedded Systems using Cadence SMV. *Conference on Electrical and Computer Engineering*, 2000 Canadian, Vol. 1, pages 179–183, 2000.

[MØl04] A. MØller. dk.brics.automaton. http://www.brics.dk/automaton, 2004

[MP11] José Vander Meulen and Charles Pecheur. Milestones: A Model Checker Combining Symbolic Model Checking and Partial Order Reduction. *NASA Formal Methods, Lecture Notes in Computer Science, Volume 6617*, pages 525–531, 2011.

[Nau64] P. Naur. Revised report on the algorithmic language algol 60. Communications of the ACM, Vol. 6(1), 1963. A/S Regnecentralen, Copenhagen, pages 1–43, 1964.

[NSLD11] Truong Khanh Nguyen, Jun Sun, Yang Liu, Jin Song Dong. A model checking framework for hierarchical systems, 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 633 - 636, 2011

[NuSMV] NuSMV: A new symbolic model checker. http://nusmv.fbk.eu. NuSMV User Manual. http://nusmv.irst.itc.it/NuSMV/userman/index-v2.html

[OM03] D. Owen, T. Menzies, Lurch: a Lightweight Alternative to Model Checking, In *Software Engineering and Knowledge Engineering* (SEKE), pp. 158-165, 2003

[OracJav] Java™ 2 Platform Standard Edition 5.0 API Specification. http://docs.oracle.com/javase/1.5.0/docs/api

[Ora13] http://www.infoworld.com/d/data-management/oracles-ellison-promises-ungodly-database-speed-new-in-memory-option-227290

[Parr07] Terence Parr. The Definitive ANTLR Reference: Building Domain-Specific Languages. Version: 2007-3-20.

[PRISM] http://www.prismmodelchecker.org/, 2013

[RB03] RuleBase: Formal Verification Tool, Version 1.4.3. Verification Technologies Group, IBM Haifa Research Laboratories, January 2003. https://www.research.ibm.com/haifa/projects/verification/RB_Homepage/

[RKSMH00] T. Rus, R. Kooima, R. Soricut, S. Munteanu and J. Hunsaker. TICS: A Component-Based Language Processing Environment. *CiteSeerX*, pages 1–10, 2000. ftp://ftp.cs.uiowa.edu/pub/rus/components2.ps

[RKW99] T. Rus, R. Kooima and E. Van Wyk. Semantics specification in an algebraic compiler. *CiteSeerX*, pages 1–37, 1999. ftp://ftp.cs.uiowa.edu/pub/rus/semspec.ps

[Roz11] K.Y. Rozier. Survey: Linear Temporal Logic Symbolic Model Checking, *Computer Science Review,* Volume 5 Issue 2, pages 163–203, 2011.

[RP97] T. Rus and S.V. Pemmaraju. Using Graph Coloring in an Algebraic Compiler. *Acta Informatica*. Vol. 34, No. 3, pages 191–209, 1997.

[Rua08] J. Ruan, Reasoning about Time, Action and Knowledge in Multi-Agent Systems, Ph.D. Thesis, University of Liverpool, http://ac.jiruan.net/thesis/, 2008

[Rus02] T. Rus. A Unified Language Processing Methodology. *Theoretical Computer Science 281*, pages 499–536, 2002.

[Rus83] Teodor Rus. Mecanisme formale pentru specificaţia limbajelor. *Editura Academiei*, Bucureşti, pages 1–475, 1983.

[Rus88] T. Rus. Parsing languages by pattern matching. *IEEE Transactions on Software Engineering*, Vol. 14(4), pages 498–510, 1988.

[Rus91] T. Rus. Algebraic construction of compilers. *Theoretical computer Science*, Vol. 90, pages 271–308, 1991.

[Rus98] T. Rus. Algebraic processing of programming languages. *Theoretical Computer Science,* Vol. 199(1), pages 105–143, 1998.

[RW96] T. Rus and E.V. Wyk. Algebraic Implementation of model checking. *In third AMAST Workshop on Real-Time Systems*, pages 267–279, 1996.

[RWH02] T. Rus, E. Van Wyk and T. Halverson. Generating model checkers from algebraic specifications. *Springer, Formal Methods în System Design*. Vol. 20, Issue 3, pages 249–284, 2002.

[SB12] Laura Florentina Stoica and Florian Mircea Boian. Algebraic approach to implementing an ATL model checker. *STUDIA UNIV. BABEŞ BOLYAI, INFORMATICA*, Cluj-Napoca, Romania. Volume LVII, Number 2, pages 73–82, 2012.

[SBS13] Laura Florentina Stoica, Florian Mircea Boian and Florin Stoica. A Distributed CTL Model Checker. *Proceeding of 10th International Conference on e-Business, Reykjavik Iceland*, paper 33, pages: 379-386, 29-31 July, 2013.

[SC10] F. Stoica and L. F. Cacovean. Interoperability Issues in Accessing Databases through Web Services. *Proceedings of the 11th WSEAS International Conference on Evolutionary Computing (EC '10)*. Iaşi, Romania, pages 279–284, 2010.

[Sch03] Klaus Schneider. Verification of Reactive Systems: Formal Methods and Algorithms. Springer Verlag Berlin Heidelberg, First Edition, pages 1-600, 2003

[Sch04] Klaus Schneider. Verification of reactive systems: formal methods and algorithms. *Springer*, page 45, 2004. ISBN 9783540002963. http://www.amazon.com/Verification-Reactive-Systems-Algorithms-Theoretical/dp/3642055559

[Sha06] Anand Sharma. Theory of Automata and Formal Languages. LAXMI Publication (P) LTD. Second Edition, pages 1-522, 2006.

[Som12] F. Somenzi. CUDD: CU decision diagram package - release 2.5.0., http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html, 2012

[SS11] Laura Florentina Stoica and Florin Stoica. Considerations about the implementation of an ATL model checker. *Second International Conference on Modelling and Development of Intelligent Systems, MDIS*. Sibiu, Romania, pages 170–179, 2011.

[SS13] Florin Stoica and Laura Stoica. Building a new CTL model checker using Web Services. *Proceeding The 21th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2013)*, At Split-Primosten, Croatia, 18-20 September, 2013.

[SSB13] Laura Florentina Stoica, Florin Stoica and Florian Mircea Boian. Using ATL model checking in agent-based applications. *Proceeding of Third International Conference on Modelling and Development of Intelligent Systems, Sibiu*, Romania, 10 –12 October, pages 127-135, 2013.

[SSS12] L.F. Stoica, F. Stoica and D. Simian. Client/Server Implementation of an ATL Model Checker Using Web Services. *Proceedings of the 16th WSEAS International Conference on Computers*, Kos Island, Greece, pages 359–364, July 14-17, 2012. ISBN: 978-1-61804-109-8.

[SW96] Markus Stumptner and Franz Wotawa. Model-Based Program Debugging and Repair. *CiteSeerX*, pages 155–160, 1996.

[Tab95] Deian Tabakov. *Experimental Evaluation of Explicit and Symbolic Automata-Theoretic Algorithms*, Master of Science thesis, Rice University, Texas, 2005.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, 1986.

[Wag05] F. Wagner. Moore or Mealy model?. Pages 1–7, April 2005. http://www.stateworks.com/active/download/TN10-Moore-Or-Mealy-Model.pdf

[Wir90] M. Wirsing. Algebraic specification. *In Handbook of theoretical Computer Science*, The Mit Press/Elsevie, Volume B, pages 677–788, 1990.

[Win84] WING, J.M. Helping specifiers evaluate their specifications. In *Proceedings of the 2nd Software Engineering Conference*. AFCET, pages 179–191, 1984.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages, an introduction. Foundations of Computing*. The MIT Press, 1993.

[WV97] J. Wing and M. Vaziri-Farahani. A case study in Model checking software systems. *Science of ComputerProgramming*, Vol 28, pages 273–299, 1997.

[WWQ05] Andy Ju An Wang, Kai Qian Andy Ju An Wang and Kai Qian. Component-oriented programming, John Wiley and Sons. Pages 1–333, 2005. ISBN: 0471644463, 9780471644460.

[Wyk00] E.V. Wyk. Specificaion Languages in Algebraic Compiler. *CitiSeerX*, pages 1–38, 2000. *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.5593&rep=rep1&type=pdf*

[Wyk98] E. Van Wyk. Semantic Processing by Macro Processors. *PhD Thesis*, *The University of Iowa, Iowa City, Iowa, CiteSeerX*, pages 1–167, 1998.

[ZD08] Yan Zhang and Yulin Ding. CTL Model Update for System Modifications. *Journal of Artificial Intelligence Research* 31, pages 113–155, 2008.