UNIVERSITATEA „BABEŞ-BOLYAI” CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

Laura Florentina Cacovean

Verificarea formală a modelelor pentru sisteme cu evenimente discrete

REZUMATUL TEZEI DE DOCTORAT

Conducător ştiinţific:
Prof. univ. dr. Florian Mircea Boian

2014

Teza conține următoarele capitole[1]:

1 Introducere
1.1. Abordări actuale în verificarea modelor
1.2. Verificare simbolică versus verificare explicită
1.3. Instrumente remarcabile pentru verificarea de modele CTL
1.4. Instrumente remarcabile pentru verificarea de modele ATL
1.5. Motivaţie şi obiective
1.6. Direcţiile de cercetare din cadrul tezei
1.7. Structura tezei
1.8. Contribuţii originale ale tezei

2 Concepte teoretice fundamentale
2.1. Structuri Kripke. Logici temporale
2.2. Conceptele algebrice fundamentale
2.2.1. Sigma algebre
2.2.2. Sigma limbaje
2.3. Concluzii

3 Dispozitiv de verificare a modelelor CTL folosind metodologia algebrică
3.1. Dispozitivul de verificare a unui model
3.1.1. Verificarea unui model CTL
3.2. Descrierea algebrică a unui verificator de modele CTL
3.2.1. Structura algebrică a limbajului CTL
3.2.2. Descrierea algebrică a verificatorului de modele CTL
3.3. Algoritmul pentru verificarea formulelor CTL
3.4. Proiectarea algebrică a verificatorului de modele CTL
3.4.1. Specificaţia algebrică a unui limbaj independent de context
3.4.2. Implementarea compilatorului algebric prin acţiuni semantice
3.5. Gramatica de specificare a $\sum$–limbajului L_{ctl}
3.5.1. Specificarea acţiunilor semantice. Generarea automată a verificatorului de modele CTL
3.5.2. Studiu de caz - Excluderea mutuală a două procese
3.6. Concluzii

4 Verificarea modelelor CTL prin gramatici atributive
4.1. Dezvoltarea verificatoarelor de modele prin compilatoare algebrice bazate pe macroprocesare. Soluţii alternative.
4.2. ANother Tool for Language Recognition
4.2.1. Generare de cod cu ANTLR
4.2.2. Eliminarea nedeterminismului
4.2.3. Acţiuni semantice
4.3. Gramatici atributive
4.4. Definirea semanticii CTL cu ajutorul teoremelor de punct fix

[1] În cadrul acestui rezumat nu sunt detaliate toate secțiunile capitolelor

Publicații conexe tezei de doctorat

Rezultatele cercetării și contribuțiile originale prezentate în teză au fost publicate în proceedings-urile conferințelor internaționale la care am participat, în revista Studia Univ. Babeş Bolyai – Informatica sau se află sub recenzie la diverse jurnale ISI, clasificate CNATDCU.

Lucrări publicate (3 conferinţe categoria B, o conferinţă categoria C, 3 conferinţe categoria D, un jurnal categoria D):

1. **Laura Florentina Stoica**, Florian Mircea Boian and Florin Stoica. *A Distributed CTL Model Checker*. Proceeding of 10th International Conference on e-Business, ICE-B 2013, Reykjavik Iceland, paper 33, pg. 379-386, 29-31 July, **2013**.
   Indexată *ISI Thomson*. **Conferință clasificată CNATDCU, categoria B**. [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.
   Indexată *ISI Thomson*. **Conferință clasificată CNATDCU, categoria B**. [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. **Conferință clasificată CNATDCU, categoria D**. [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.**
   Revista clasificată CNATDCU, categoria D. [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*. **Conferință clasificată CNATDCU, categoria C.** [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*. **Conferință clasificată CNATDCU, categoria D.** [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*. **Conferință clasificată CNATDCU, categoria D.** [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*. **Conferință clasificată CNATDCU, categoria B.** [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)

Lucrări aflate sub recenzie la jurnale ISI, clasificate 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), **jurnal clasificat CNATDCU, categoria B.**

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/), **jurnal clasificat CNATDCU, categoria C.**

1. Introducere

1.1. Abordări actuale în verificarea modelor

În prezent se manifestă o creştere continuă a complexităţii sistemelor, tendinţă care este susţinută şi accelerată de progresul tehnologic în domeniul arhitecturilor hardware, a produselor software şi a sistemelor de comunicații. O provocare deosebită adresată cercetătorilor în domeniului informaticii este aceea de a furniza formalisme, instrumente şi tehnici care să permită proiectarea de sisteme fiabile, eficiente şi fără erori, în pofida complexităţii acestora.

Testarea şi simularea întăresc încrederea în corectitudinea implementării unui sistem software, dar nu pot dovedi faptul că toate erorile au fost eliminate. Mai mult, testarea nu este nici exhaustivă şi nici foarte eficientă în cazul sistemelor software concurente, care sunt mult mai complexe decât sistemele secvenţiale. Astfel, s-a manifestat un interes crescut pentru algoritmi şi tehnici care să permită demonstrarea faptului că un program satisface anumite proprietăţi. Procesul de specificare şi demonstrare a proprietăţilor sistemelor software poartă numele de verificarea programelor.

Verificarea unui sistem software implică confirmarea faptului că sistemul în cauză se comportă conform modului în care a fost proiectat. Validarea proiectării unui sistem implică verificarea faptului că sistemul respectiv satisface cerinţele de sistem. Aceste două sarcini, verificarea sistemelor şi validarea proiectării pot fi realizate complet, fiabil şi riguros prin utilizarea metodelor formale bazate pe modele, cum ar fi verificarea de modele [Roz11].

O abordare extrem de atractivă care vizează corectitudinea sistemelor hardware şi software constă în construirea şi verificarea modelelor care surprind comportamentul sistemelor respective.

Verificarea de modele este în mod particular foarte potrivită pentru verificarea automată a sistemelor cu stări finite, hardware sau software.

Verificarea unui model reprezintă o tehnică formală care stabileşte fără echivoc, în mod automat şi exhaustiv, dacă modelul respectiv satisface o anumită specificaţie. În general, aceasta presupune o inspectare sistematică a tuturor stărilor modelului. Altfel spus, verificarea modelelor este o tehnică pentru verificarea automată a corectitudinii sistemelor cu stări finite.

În verificarea formală a modelelor, pentru definirea specificaţiilor pe care trebuie să le satisfacă sistemele hardware sau software se utilizează logicile temporale.

Câteva verificatoare de modele consacrate, care şi-au câştigat atributul de „state of the art” prin prisma performaţelor, gradului de utilizare şi a facilităţilor oferite sunt prezentate în continuare.

1.1.1. Instrumente remarcabile pentru verificarea de modele CTL

Dintre sistemele de verificare a modelelor CTL, două instrumente sunt remarcabile în opinia noastră din punct de vedere al facilităţilor oferite şi pot fi utilizate cu succes în verificarea modelelor CTL: NuSMV şi respectiv RuleBase.

Sistemul NuSMV este rezultatul unui proiect comun derulat între *Carnegie Mellon University* (CMU) şi *Istituto per la Ricerca Scientifica e Tecnologica* (IRST) şi este practic succesorul verificatorului de modele CMU SMV. Modelarea sistemelor în NuSMV se realizează prin intermediul unui limbaj care permite descrierea maşinilor cu stări finite - *Finite State Machines* (FSM). Limbajul facilitează descrieri modulare, ierarhice şi permite definirea componentelor reutilizabile. Principalul scop al unui model NuSMV este descrierea relaţiilor de tranziţie în cadrul FSM, relaţii care definesc evoluţiile valide ale stării maşinii cu stări finite [CCGPRST02].

Sistemul RuleBase a fost implementat de IBM şi se bazează tot pe CMU SMV, fiind de fapt o versiune îmbunătăţită a acestuia, pentru a face posibilă aplicarea sa în aplicaţii industriale. Pentru a face specificaţiile CTL mai simple, RuleBase utilizează un limbaj propriu numit *Sugar*, special proiectat pentru proiectanţii hardware, pentru care este dificil de înţeles complexitatea semanticii CTL [RB03]. Eforturile IBM s-au direcţionat în crearea unor algoritmi care adresează optimizări ale diagramelor DDB utilizate în reprezentarea internă a modelelor verificate, pentru a face posibilă verificarea unor modele de mari dimensiuni.

1.1.2. Instrumente remarcabile pentru verificarea de modele ATL

În [AHMQRT98] este prezentat un mediu de verificare numit MOCHA utilizat în verificarea modulară a sistemelor eterogene. Trebuie subliniat că MOCHA a fost implementat chiar de către autorii care au definit logica ATL (*R. Alur, T.A. Henzinger*). Limbajul de intrare pentru MOCHA este bazat pe conceptul de module reactive. Modulele reactive furnizează un liant semantic care permite încapsularea formală şi interacţiunea unor componente cu caracteristici diferite [AHMQRT98]. O definire formală a modulelor reactive se găseşte în [AH96].

În lucrarea [LR06] este descris sistemul MCMAS, un verificator simbolic de modele proiectat special pentru scenarii şi specificaţii bazate pe agenţi. MCMAS suportă specificaţii bazate pe CTL şi ATL, logică epistemică şi modalităţi deontice pentru corectitudine. MCMAS implementează algoritmi bazaţi pe diagrame de decizie binare ordonate (*Ordered Binary Decision Diagrams*) optimizate pentru sisteme interpretate şi suportă generarea de contraexemple şi execuţie interactivă (atât în mod explicit cât şi în mod simbolic). MCMAS a fost utilizat în diverse scenarii, care includ servicii Web, diagnoză şi securitate. MCMAS utilizează un limbaj de programare dedicat numit ISPL (*Interpreted Systems Programming Language*) pentru descrierea modelelor. Un program ISPL poate descrie complet un sistem multi-agent (compus din agenţi şi din mediu).

1.2. Motivaţie şi obiective

Scopul cercetărilor noastre în domeniul verificării de modele ale sistemelor închise, respectiv deschise este acela de a dezvolta instrumente fiabile, uşor de întreţinut, scalabile, care să încurajeze şi să îmbunătăţească aplicabilitatea verificării de modele CTL, respectiv ATL în proiectarea sistemelor software de uz general.

Sistemele concurente sunt asincrone deoarece execuţia anumitor componente se poate efectua pe procesoare diferite sau poate fi intercalată de către planificatorul sistemului de operare. Având în vedere aspectele considerate anterior, instrumentele proprii de verificare a modelelor CTL/ATL se bazează pe tehnica de enumerare explicită a stărilor.

În prezent, cea mai presantă provocare în verificarea de modele este scalabilitatea [Roz11].

Pentru ca un verificator de modele să poată fi utilizat cu succes în aplicaţii reale trebuie să fie eficient, scalabilitatea sa urmând a fi evaluată în raport cu următoarele criterii:

- Dimensiunea modelelor pe care este capabil sa le gestioneze;
- Timpul şi spaţiul de memorie necesar verificării modelelor complexe.

Pentru a adresa problema exploziei spaţiului de stări, verificatorul nostru de modele CTL se bazează pe structuri de date eficiente utilizate în reprezentarea internă a modelelor, furnizate de către biblioteca GraphStream [GrStr12], iar verificatorul de modele ATL utilizează exclusiv sisteme de gestiune a bazelor de date pentru memorarea reprezentărilor structurilor de joc concurente. Pe partea client, implementarea C# utilizează structuri de date proprii, inspirate din [Ebe87], pentru memorarea multi-grafelor orientate iar bibliotecile Java se bazează pe GraphStream. Ambele instrumente utilizează tehnologia serviciilor Web având în vedere următoarele funcţionalităţi şi obiective:

- Exploatarea resurselor de calcul şi a spaţiului de memorie oferite de servere puternice;
- Oportunitatea de mărire a performanţelor prin extensii care vizează paralelizarea procesului de verificare a modelelor;
- Instalarea extrem de simplă a produselor software pe partea de client, verificatoarele de modele devenind astfel imediat disponibile, fără a necesita vreo configurare;
- Utilizarea instrumentelor de verificare în medii educaţionale sau în scop de cercetare, construirea modelelor fiind realizată interactiv sau programatic pe partea client, iar verificarea modelelor respective fiind efectuată prin invocarea transparentă a serviciilor Web corespunzătoare;
- Serviciile Web sunt disponibile online (https://mcheck-useit.rhcloud.com, http://use-it.ro), dar software-ul este disponibil şi pentru instalare în reţele locale.

1.3. Structura tezei

Capitolul 1 prezintă stadiul actual al cercetărilor în domeniul utilizării logicilor temporale în verificarea sistemelor şi o analiză a avantajelor / dezavantajelor verificării explicite în raport cu verificarea simbolică. De asemenea, se analizează succint câteva instrumente consacrate pentru verificarea de modele CTL, respectiv ATL. În continuare, se prezintă motivaţia studiului întreprins, obiectivele şi direcţiile de cercetare din cadrul tezei.

În **capitolul 2** se descriu conceptele algebrice fundamentale utilizate în cadrul metodologiei algebrice de dezvoltare a compilatoarelor algebrice în general, şi în implementarea verificatoarelor de modele în particular.

Capitolul 3 constituie o descriere amplă a tuturor aspectelor care trebuie analizate în utilizarea metodologiei algebrice pentru implementarea unui instrument software capabil să verifice în mod automat sisteme modelate cu ajutorul logicilor temporale. Acest capitol începe printr-o prezentare generală a conceptului de verificator de modele bazat pe logici temporale precum şi a etapelor din cadrul procesului de verificare a unui model. În continuare este descrisă metodologia algebrică propusă de Rus [Rus91, CPBP08] pentru proiectarea compilatoarelor utilizând specificaţii algebrice şi aplicaţiile acesteia în implementarea verificatoarelor de modele CTL. Secţiunea 3.2 cuprinde aspectele teoretice ale definirii compilatorului algebric care reprezintă în fapt verificatorul de modele CTL. În secţiunea 3.3 se prezintă implementarea la nivel abstract a homomorfismului între algebrele de sintaxă ale limbajelor sursă, respectiv ţintă, care realizează practic verificarea formulelor CTL în cadrul unui model dat. Secţiunea 3.4 conţine specificaţia algebrică a unui limbaj independent de context, ce constituie premisele proiectătii verificatorului de modele CTL pornind de la gramatica independentă de context care generează limbajul fomulelor CTL (proces detaliat în secţiunea 3.4.2). Gramatica de specificare a Σ–limbajului formulelor CTL este prezentată în secţiunea 3.5, iar capitolul se încheie cu specificaţia completă a compilatorului algebric (verificatorului de modele CTL). În studiul de caz prezentat în secţiunea 3.5.2, un model CTL pentru excluderea mutuală a două procese, se urmăreşte execuţia pas cu pas a compilatorului algebric proiectat, în cadrul procesului de verificare a unor specificaţii ale sistemului modelat, exprimate ca şi formule CTL. Deşi urmează aceleaşi principii ale metodologiei algebrice propuse de Rus, verificatorul nostru de modele CTL prezintă următoarele diferenţe structurale faţă de cel prezentat în [Wyk98]: suportă sintaxa completă a formulelor CTL, respectiv toţi operatorii modali (în [Wyk98], verificatorul CTL are suport numai pentru patru operatori temporali); diferă setul de operaţii dependente de model, în soluţia noastră am optat pentru un set de operaţii care au o sintaxă asemănătoare celor utilizate în verificatorul de modele ATL, prezentat în capitolul 6; compilatorul algebric este generat cu ajutorul ANTLR, pe baza unei gramatici atributive (descrisă în capitolul 4), iar analizorul sintactic este de tip top-down, spre deosebire de cel furnizat de sistemul TICS, utilizat în [Wyk98], şi care este de tip bottom-up şi se bazează pe macroprocesare în procesul de translatare.

În **capitolul 4** este prezentat generatorul de analizoare ANTLR, pe care se bazează implementarea verificatorului nostru de modele CTL şi se justifică alegerea acestui instrument în detrimentul altora (YACC, FLEX, BISON, BYACC/J, etc). Gramaticile atributive sunt prezentate ca alternative ale dezvoltării compilatoarelor algebrice prin macroprocesare (soluţia adoptată de sistemul TICS, şi aplicată în [Wyk98]) şi se enumeră argumentele care recomandă utilizarea gramaticilor atributive ANTLR în implementarea de verificatoare de modele. Sunt prezentate concepte avansate utilizate de ANTLR în generarea translatoarelor: analiză de tip LL(k), LL(*), PEG, meta-limbaj flexibil de specificare a gramaticilor, care permite plasarea unei acţiuni semantice înainte şi/sau după specificarea unei reguli de producţie, predicate sintactice, predicate semantice, *memoizare*, automate finite de decizie cu rol de predicţie în procesul de analiză, *auto-backtracking* pentru analiza gramaticilor non-LL(*), tehnici specifice de eliminare a nederminismului. De asemenea, se prezintă modalitatea de implementare a acţiunilor semantice în ANTLR, ce reprezintă conceptul de legătură între evaluarea atributelor în cadrul gramaticii care generează limbajul formulelor CTL şi implementarea compilatorului algebric care reprezintă verificatorul de modele CTL. Astfel, o acţiune semantică asociată unei reguli de producţie reprezintă modalitatea de implementare a operaţiei derivate asociate operatorului CTL pentru care a fost definită producţia respectivă. În acelaşi timp însă, rolul acţiunii semantice este acela de a calcula valoarea atributului neterminalului care se retranscrie prin regula de producţie căreia îi este asociată.

În **capitolul 5**, pentru a face disponibilă implementarea compilatorului algebric CTL ca şi componentă reutilizabilă a instrumentului de verificare a modelelor CTL, a fost realizată publicarea acesteia ca şi serviciu Web. Compilatorul algebric CTL, încapsulat în serviciul WEB şi bazat pe codul Java generat de ANTLR pe baza unei gramatici CTL atributive originale, va realiza verificarea unei formule CTL în cadrul unui model dat, furnizând totodată semnalarea eventualelor erori lexicale/sintactice în formula pentru care se determină mulţimea sa de satisfacere. Algoritmul pentru determinarea unei strategii câştigătoare pentru jocul *X0* a fost utilizat pentru evaluarea performanţei instrumentului de verificare a modelelor CTL.

În **capitolul 6** se prezintă o extensie CTL, numită ATL (Alternating-time Temporal Logic). Logica temporală ATL stă la baza unor modele adecvate sistemelor deschise, care descriu în mod natural procesările din cadrul sistemelor multi-agent, jocurilor multi-utilizator, etc. În cadrul capitolului se arată cum metodologia algebrică poate fi utilizată la dezvoltarea dispozitivelor de verificare a modelelor ATL. În secţiunea 6.2 se defineşte în mod formal o structură de joc concurent. În secţiunea 6.3 este prezentată logica ATL împreună cu sintaxa şi semantica acesteia. Subcapitolul 6.5 cuprinde structura şi descrierea algebrică a dispozitivului de verificare a modelelor ATL. Descrierea structurii algebrice a limbajului ATL are ca punct de plecare definirea limbajul formulelor ATL ca Σ–limbaj. Metodologia algebrică de proiectare a unui verificator de modele CTL, prezentată în detaliu în capitolul 3, a fost aplicată cu succes pentru descrierea şi proiectarea algebrică a unui verificator de modele ATL. Specificarea compilatorului algebric ATL a fost detaliată în secţiunea 6.7, prin definirea sintaxei EBNF şi a acţiunilor semantice corespunzătoare producţiilor gramaticii de specificare a Σ–limbajului formulelor ATL. În secţiunile 6.9 şi 6.10 se realizează o formalizare a funcţiei *Pre()*, utilizată în toate operaţiile derivate corespunzătoare operatorilor ATL modali, folosind concepte ale Algebrei Relaţionale. Apeluri concrete ale acestei funcţii (care în terminologia metodologiei algebrice este o operaţie dependentă de model) au fost exemplificate în procesul de verificare al unor specificaţii formulate în cadrul unui model ATL original pentru problema secţiunii critice rezolvată folosind un mutex.

În **capitolul 7** se prezintă arhitectura client/server a verificatorului de modele ATL, care se bazează pe tehnologia serviciilor Web pentru a expune funcţionalitatea componentei de bază, compilatorul algebric dezvoltat în capitolul anterior. Verificatorul de modele ATL se compune dintr-o aplicaţie client care permite construirea grafică interactivă a modelelor ATL, componenta server (serviciul Web) şi biblioteci API, care permit specificarea programatică a modelelor ATL în format XML precum şi verificarea formulelor ATL prin invocarea serviciului Web, disponibile pentru limbajele C# şi Java. De asemenea, s-a realizat evaluarea performanţei verificatorului de modele în raport cu trei servere de date: MySql, SQL Server sau H2. Au fost realizate două aplicaţii ale verificatorului de modele ATL, care vizează determinarea strategiilor optime în sisteme multi-agent modelate ca structuri sincrone de jocuri concurente alternative şi respectiv validarea comportamentelor de tip FSM (Finite State Machine) ale agenţilor JADE.

Capitolul 8 cuprinde concluziile tezei şi prezintă direcţiile de cercetare viitoare, conexe domeniului abordat în cadrul lucrării.

În cadrul lucrării, tabelele şi figurile au fost numerotate cu numere consecutive, prefixate de numărul secţiunii în care apar.

1.4. Contribuţii originale ale tezei

Principalele contribuții originale ale tezei şi lucrările în care acestea au fost publicate sunt:

- Extinderea algebrei de sintaxă Sin_{ctl} a limbajului L_{ctl} prezentată în [Wyk98] cu mulţimea de operatori $\{\rightarrow, AG, AF, EG, EF\}$. Construcţia homomorfismului $T_{MC} : Sin_{ctl} \rightarrow Sin_M$ a fost de asemenea extinsă prin definirea a câte unei operaţii derivate $d_{MC}(op)$ în algebra de cuvinte Sin_M a limbajului ţintă pentru fiecare operator nou introdus [CS08, CS09, Ca09];
- Definirea a două operaţii dependente de model, $pre_\forall()$ şi respectiv $pre_\exists()$ care sunt utilizate în cadrul tuturor operaţiilor derivate asociate operatorilor CTL modali; aceasta a permis simplificarea sintaxei operaţiilor derivate din algebra Sin_M [SBS13, SS13];
- Extinderea specificaţiei algebrice a unui limbaj independent de context pentru cazul în care producţiile gramaticii care generează limbajul respectiv sunt specificate în sintaxă EBNF. Această extindere a fost necesară deoarece gramatica ANTLR de specificare a Σ–limbajului L_{ctl} utilizează sintaxa EBNF pentru anumite producţii asociate operatorilor CTL, cu scopul eliminării recursivităţii de stânga;

- Demonstrarea obţinerii denotaţiei (mulţimii de satisfacere) pentru formula CTL $AG\ f$ ca punct fix al funcţiei $g(X) = [\![f]\!] \cap pre_{\forall}(X)$;
- Analiza complexităţii algoritmului de verificare a modelelor CTL;
- Implementarea/generarea compilatorului algebric prin gramatica atributivă ANTLR de specificare a Σ–limbajului L_{ctl} [CS08, CS11];
- Expunerea funcţionalităţii compilatorului algebric prin intermediul unui serviciu Web – *CTL Checker* [CSS11];
- *CTL Designer* – Componenta client a instrumentului de verificare a modelelor CTL, aplicație cu interfaţă grafică dezvoltată în C# care permite construirea şi verificarea interactivă a modelelor CTL [CSS11];
- Proiectarea algoritmului pentru determinarea strategiei optime într-un joc de tip sincron alternativ şi utilizarea acestuia pentru evaluarea performanţei verificatorului de modele CTL [SBS13, SS13];
- Definirea limbajului formulelor ATL ca *Σ–limbaj*, definirea operaţiilor derivate şi a structurii compilatorului algebric care reprezintă verificatorul de modele ATL [SS11, SB12, SSS12];
- Implementarea compilatorului algebric ATL prin gramatica atributivă ANTLR de specificare a Σ–limbajului L_{atl} [SSS12];
- Construirea unei structuri sincrone de joc alternativ (model ATL) pentru controlul accesului la un site Web [SSS12];
- Formalizarea funcţiei *Pre*() – considerată operaţie dependentă de model în algebra de sintaxă a limbajului ţintă pentru compilatorul algebric – prin expresii de algebre relaţionale şi implementarea acesteia prin translatarea expresiilor respective în limbaj SQL;
- Publicarea dispozitivului de verificare de modele ATL ca serviciu Web – *ATL Checker* [SB12, SSS12];
- *ATL Designer* – Componenta client a instrumentului de verificare a modelelor ATL, aplicație cu interfaţă grafică dezvoltată în C#, permite construirea şi verificarea interactivă a modelelor ATL. Pentru reprezentarea internă a unui model ATL ca şi multi-graf orientat, implementarea noastră este bazată pe structuri de date adecvate grafelor dinamice [Ebe87]. Aceste structuri au fost adaptate pentru limbajul C# şi apoi extinse pentru necesităţile de reprezentare a structurilor de joc concurente [SB12, SSS12];
- Dezvoltarea unor interfeţe de programare API – ATL Library – pentru construirea programatică a modelelor CTL/ ATL de mari dimensiuni;
- Construirea şi verificarea unui model ATL pentru două procese concurente care doresc să intre într-o secţiune critică. Soluţia noastră îmbunătăţeşte modelul CTL clasic deoarece suportă o concurenţă reală: două procese pot solicita simultan intrarea în secţiunea critică, iar accesul lor este restricţionat folosind un mutex gestionat de sistemul de operare, reprezentat în modelul nostru de un agent;
- Modificarea modelului ATL propus de Alur [AHK02] pentru a fi posibilă reprezentarea mutărilor agenţilor prin simboluri arbitrare (în modelul original mutările agenţilor erau reprezentate prin numere naturale) [SSB13, SB12];

- Proiectarea algoritmului pentru determinarea strategiei optime într-un joc de tip sincron alternativ şi utilizarea acestuia pentru evaluarea performanţei verificatorului de modele ATL;
- Dezvoltarea unei tehnici de validare a comportamentelor de tip FSM (Finite State Machine) ale agenţilor JADE. Soluţia propusă se bazează pe versiunea Java a componentei ATL Library, şi permite verificarea la momentul execuţiei agenţilor a unor specificaţii exprimate prin formule ATL. Modelele ATL sunt construite în mod automat, odată cu definirea maşinilor cu stări finite ale comportamentelor agenţilor JADE;
- Implementarea suportului verificatorului de modele ATL pentru trei servere de baze de date: MySQL, SQL Server şi H2.

1.5. Cuvinte cheie:

Sistem software, verificare formală, logică temporală, verificator de modele, logica CTL, verificarea de modele CTL, structură Kripke, model CTL, logica ATL, verificarea de modele ATL, model ATL, structură de joc concurent, sistem multi-agent, compilator algebric, ANTLR, gramatică indpendentă de context, gramatică atributivă ANTLR, serviciu Web, Algebră relațională, SQL.

2. Concepte teoretice fundamentale

În cadrul acestui capitol se descriu conceptele algebrice fundamentale utilizate în cadrul metodologiei algebrice, în general de dezvoltare a compilatoarelor algebrice, şi în particular în implementarea verificatoarelor de modele.

Fundamentele algebrice sunt necesare pentru a înţelege conceptele teoretice pe care se bazează compilatoarele algebrice. Teoria algebrică este baza instrumentelor generatoare de compilatoare. În cadrul capitolului sunt prezentate conceptele de limbaj şi limbaj de programare. Procesarea limbajelor implică dezvoltarea de modele matematice ale acestora. Proprietăţile de transformare ale textelor bine formate ale unui limbaj sursă în elemente aparţinând unui alt limbaj ţintă conservă semnificaţia textului original. Compilarea limbajelor de programare sau translatarea unui limbaj în alt limbaj de nivel înalt sau în limbajul natural sunt exemple de astfel de transformări. Dificultăţile computaţionale în proiectarea translatării unui limbaj sunt legate de natura universului de discuţie al limbajului respectiv.

De asemenea sunt prezentate fundamentele metodologiei algebrice unde limbajele sunt reprezentate folosind *sigma algebre*. Metodologia algebrică este o metodologie de compoziţie, deoarece un limbaj este specificat de o mulţime finită de reguli care determină modul de construire a limbajului folosind componentele sale.

În metodologia algebrică limbajele sunt reprezentate folosind sigma algebre, iar compilatoarele sunt specificate ca *homomorfisme generalizate* care înglobează limbajul sursă în limbaj ţintă. Prin translatarea elementelor din limbajul sursă se ajunge la reprezentări ale limbajului ţintă.

Procedura de calcul a homomorfismelor generalizate între algebre nesimilare se poate utiliza în cadrul compilatoarelor algebrice pentru a defini translatoare între limbaje care au fost definite utilizând algebre nesimilare.

Metodologia expusă poate fi utilizată în implementarea *verificatoarelor de modele* ca şi compilatoare algebrice, în cadrul cărora limbajele sursă sunt reprezentate de limbaje ale logicilor temporale care definesc sintaxa şi semantica formulelor logice temporale, iar limbajul ţintă este limbajul mulțimilor de stări care satisfac formulele logice în cadrul unor modele specifice.

3. Dispozitiv de verificare a modelelor CTL folosind metodologia algebrică

Algoritmii de verificare a unui model sunt în prezent utilizaţi ca tehnici de verificare a sistemelor în medii de programare diferite, care includ fenomene în timp real şi în programare paralelă. Diversitatea sistemelor şi a mediilor impun o diversitate de logici şi de algoritmi. S-au construit până acum instrumente care îi pot ajuta pe programatori sau pe practicieni să utilizeze variate logici temporale. Instrumentele dau utilizatorului posibilitatea să verifice şi să extindă o logică temporală. Acesta are posibilitatea să utilizeze un dispozitiv de verificare a unui model adecvat domeniului problemei. Printre cercetătorii care au dezvoltat astfel de instrumente se află şi Rus [RKSMH00], care a dezvoltat un set de instrumente ce furnizează aceste capabilităţi de specificaţie prin plasarea problemei verificării modelului într-un cadru algebric. Metodologia algebrică furnizează o "platformă de test logic temporal", care permite uşor prototipizarea unui dispozitiv de verificare de model. Instrumentele pe care le vom folosi pentru metodologia algebrică vor genera un algoritm de verificare a modelului, printr-o reprezentare algebrică folosind logici temporale. Acest algoritm constă într-un proces de mapare a unei algebre sursă pe algebra ţintă, realizat prin operaţii derivate implementate în cadrul unui compilator algebric.

Logica temporală se foloseşte pentru a exprima proprietăţile unui sistem atât calitativ [CES86, BCM92, McM93] cât şi cantitativ [EMSS91, CC94]. Dispozitivele de verificare de modele sunt la rândul lor instrumente ce pot fi folosite pentru a verifica dacă un sistem dat satisface o formulă de logică temporală dată. Sistemul verificat poate fi un sistem fizic sau un program concurent în timp real al cărui comportament este descris de *modelul Kripke* [Kri93].

Este extrem de utilă dezvoltarea unor instrumente flexibile pentru verificarea de modele, care să permită (re)generarea algoritmilor de verificare în cazul schimbării unor specificaţii (spre exemplu, adaugarea de proprietăţi ale logicii temporale, etc.). Această problemă este similară cu cea întâlnită de cercetătorii limbajelor de programare care doresc să proiecteze şi să experimenteze rapid şi uşor noi construcţii de limbaj, fără a rescrie compilatoarele atunci când se adaugă funcţii noi în cadrul limbajului. Instrumentele de generare a compilatoarelor oferă o soluţie parţială pentru aceste cerinţe, deoarece pot genera automat noi compilatoare şi noi componente din specificaţii variabile. Rus a dezvoltat o metodologie şi un set de instrumente pentru generarea de compilatoare utilizând specificaţii algebrice ale limbajelor (sursă respectiv ţintă), care pot fi utilizate cu succes în implementarea verificatoarelor de modele pentru logici temporale.

În cadrul acestui capitol s-a prezentat metodologia propusă de Rus pentru proiectarea compilatoarelor utilizând specificaţii algebrice şi aplicaţiile acesteia în implementarea verificatoarelor de modele CTL.

CTL permite descrierea şi modelarea sistemelor închise, al căror comportament este complet determinat de către stările sistemului. Modelele CTL sunt reprezentate de către structuri Kripke şi sunt construite pentru a reprezenta comportamentul sistemelor închise.

Verificarea modelelor este una dintre cele mai utilizate tehnologii aplicate în verificarea de sisteme automate.

Formal, un *model M* este definit ca un graf orientat $M = (S, Rel, P)$ unde S este o mulţime finită de stări (numite şi noduri), Rel este o mulţime finită de arce orientate şi $P:S\rightarrow 2^{AP}$ este funcţia de etichetare ce etichetează fiecare nod cu propoziţii atomice din mulţimea AP. Rel este o relaţie binară totală pe S, $Rel\subseteq S\times S$ astfel încât fiecare stare din graful M are cel puţin un succesor. Pentru fiecare $s\in S$, există un $s'\in S$ astfel încât perechea $(s,s')\in Rel$. Notaţia $succ(s)=\{s'\in S|(s,s')\in Rel\}$ se va folosi pentru reprezentarea mulţimii stărilor succesoare stării s. Funcţia de etichetare P mapează stările din S cu propoziţiile atomice din $AP=\{ap_1,ap_2,...,ap_n\}$ care sunt *adevărate* în stările respective [CES86]. În verificatorul de modele CTL vom utiliza şi funcţia $P':AP\rightarrow 2^S$, care asociază fiecărei propoziţii atomice mulţimea stărilor etichetate cu propoziţia atomică respectivă, astfel că $P'(ap)=\{s \in S \mid ap \in P(s)\}$, $\forall ap \in AP$.

Un algoritm de verificare a unui model CTL a fost dezvoltat de către *Clarke* [CES86, CGL96]. Ca date de intrare se dau formula CTL notată cu f şi modelul CTL notat cu M, iar apoi toate stările s ale lui M sunt etichetate cu toate subformulele din f care sunt satisfăcute de s. Algoritmul converteşte formula f într-o formă prefixată şi apoi începând de la sfârşitul formulei identifică subformulele lui f.

Un verificator de modele CTL constă într-un compilator algebric $\mathcal{C}:L_s\rightarrow L_t$ unde limbajul sursă este limbajul formulelor CTL, iar limbajul ţintă este limbajul care descrie mulţimea de noduri (stări) ale modelelor CTL (reprezentate prin structuri Kripke) în care formulele respective sunt satisfăcute.

Într-un compilator algebric $\mathcal{C}:L_s\rightarrow L_t$, limbajul sursă L_s şi limbajul ţintă L_t sunt Σ – limbaje definite în general prin algebre care nu sunt similare.

Construirea efectivă a compilatorului algebric necesită implementarea unei proceduri de calcul a homomorfismului generalizat care asociază în mod unic oricărei construcţii sintactice a limbajului sursă o construcţie sintactică a limbajului ţintă. Astfel, compilatorul algebric $\mathcal{C}$ translatează o formulă CTL corespunzătoare modelului M, la mulţimea de noduri S', peste care formula *CTL* este satisfăcută. Prin urmare, $\mathcal{C}(f)=S'$ unde f este formula CTL de verificat şi $S'=\{s\in S|\ (M,s) \models f\}$ [CS08, CS09].

Pentru a defini limbajul formulelor CTL ca un *Σ–limbaj*, trebuie definită mai întâi schema operator a acestuia, reprezentată ca un triplet de forma $\Sigma_{ctl}=(Car_{ctl}, O_{ctl}, \sigma_{ctl})$. Car_{ctl} este o mulţime cu un singur element care reprezintă mulţimea formulelor CTL, $Car_{ctl}=\{\mathcal{F}\}$ [RW96, Wyk00]. O_{ctl} este o mulţime finită de nume de operatori, unde $O_{ctl} = \{\top, \bot, \neg, \wedge, \vee, \rightarrow, AX, EX, AU, EU, EF, AF, EG, AG\}$ [CS08, Ca09]. Funcţia σ_{ctl} defineşte signatura operatorilor din O_{ctl}.

Σ_{ctl} – limbajul L_{ctl} se defineşte ca un triplet de forma $L_{ctl} = (Sem_{ctl}, Sin_{ctl}, \mathcal{L}_{ctl} : Sem_{ctl} \rightarrow Sin_{ctl})$ unde Sin_{ctl} este algebra de cuvinte cu schema operator Σ_{ctl} generată de operaţiile din O_{ctl} şi de o mulţime finită de variabile numite propoziţii atomice, AP. Sem_{ctl} este algebra de semantică a modelului CTL definită peste mulţimile de stări în care formulele CTL corespunzătoare unui model M sunt satisfăcute.

Funcţia de învăţare $\mathcal{L}_{ctl}$ mapează o mulţime de stări ale modelului cu formulele CTL satisfăcute de mulţimea de stări respectivă (dacă există astfel de formule). Astfel, $\mathcal{L}_{ctl}$ asociază mulţimi de satisfacere cu formulele CTL satisfăcute de către aceste mulţimi.

Funcţia de evaluare $\mathcal{E}_{ctl} : Sin_{ctl} \rightarrow Sem_{ctl}$ este un homomorfism care evaluează formulele CTL, asociind în mod unic fiecărei formule mulţimea de stări ale modelului în care formula respectivă este satisfăcută (mulţimea de satisfacere a formulei respective).

Algebra de sintaxă Sin_{ctl} este independentă de orice model CTL, dar algebra de semantică Sem_{ctl} este dependentă de un model dat, datorită faptului că regulile de construcţie a mulţimii suport a algebrei de cuvinte Sin_{ctl} (regulile de formare a formulelor CTL corecte sintactic) sunt independente de modelul considerat, dar semnificaţia formulelor respective este dependentă de modelul respectiv.

Astfel, pentru modelul $M=(S,Rel,P:S\rightarrow 2^{AP})$, mulţimea suport a algebrei de semantică Sem_{ctl} este constituită din mulţimile de stări $\subseteq S$ care satisfac formulele din $Sin_{\mathcal{F}}$, prin urmare $Sem_{\mathcal{F}}\subseteq 2^S$.

Algebra de semantică Sem_{ctl} are aceeaşi schemă operator Σ_{ctl} ca şi algebra de sintaxă Sin_{ctl} (cele două algebre sunt similare). Deoarece mulţimile suport ale celor două algebre sunt diferite, operatorii din algebrele respective, deşi au aceeaşi signatură, sunt diferiţi. Astfel, fiecărui operator din Sin_{ctl} îi corespunde un operator în Sem_{ctl}, dar vom utiliza simboluri diferite pentru a nota operatorii respectivi.

Operatorii din Sem_{ctl} care corespund numelor de operatori $\{\top, \bot, \neg, \wedge, \vee, \rightarrow, AX, EX, AU, EU, AG, AF, EG, EF\}$ sunt denumiţi astfel: $\{S, \emptyset, C, \cap, \cup, S_I, S_{AX}, S_{EX}, S_{AU}, S_{EU}, S_{AG}, S_{AF}, S_{EG}, S_{EF}\}$.

În cadrul acestui capitol am definit operaţiile dependente de model, $pre_\forall()$ şi respectiv $pre_\exists()$ şi le-am utilizat în cadrul tuturor operaţiilor derivate asociate operatorilor CTL modali; aceasta a permis simplificarea sintaxei operaţiilor derivate din algebra Sin_M [SBS13, SS13];

$$pre_\forall(Z) = \{s \in S \mid succ(s) \subseteq Z\}, \text{ respectiv}$$

$$pre_\exists(Z) = \{s \in S \mid succ(s) \cap Z \neq \phi\}, \forall Z\subseteq S,$$

unde $succ(s)=\{s'\in S|(s,s')\in Rel\}$ reprezintă mulţimea stărilor succesoare stării s în M.

În cadrul acestui capitol algebra de sintaxă Sin_{ctl} a limbajului L_{ctl} prezentată în [Wyk98] a fost extinsă cu mulţimea de operatori $\{\rightarrow, AG, AF, EG, EF\}$. S-a definit câte o operaţie derivate în algebra de cuvinte Sin_M a limbajului ţintă pentru fiecare operator nou introdus [CS08, CS09, Ca09];

Definiţie 3.1. O *algebră de cuvinte* sau de *termeni* pentru o schemă operator $\Sigma=\langle Car,O,\sigma\rangle$ şi o familie de mulţimi de variabile $X=\{X_c\}_{c\in Car}$ este o *Σ-algebră* notată $Cuv_\Sigma(X)$, care constă dintr-o familie de mulţimi suport $\{Cuv_c\}_{c\in Car}$, un set de funcţii $\{op^{Cuv}|op\in O\}$ definite pe aceste mulţimi şi este generată de către variabilele X conform următoarelor reguli:

r1. dacă $x\in X_c$, atunci variabila x este un cuvânt în Cuv_c

r2. dacă $op\in O$ este un nume de funcţie cu aritatea 0, având signatura $\sigma(op)=\emptyset\rightarrow c$, atunci op este un cuvânt în Cuv_c, iar funcţia corespunzătoare este definită astfel: $op^{Cuv}()=op$.

r3. dacă $op\in O$ este un nume de funcţie cu signatura $\sigma(op)=c_1\times c_2\times...\times c_n\rightarrow c$, şi pentru orice i, $i=\overline{1,n}$, $cuv_i\in Cuv_{c_i}$, atunci cuvântul $opcuv_1cuv_2...cuv_n$ este un cuvânt în Cuv_c, funcţia corespunzătoare fiind definită astfel: $op^{Cuv}(cuv_1,cuv_2,...,cuv_n)=opcuv_1cuv_2...cuv_n$ unde $cuv_i\in Cuv_{c_i}$, $i=\overline{1,n}$.

Un verificator de modele CTL poate fi definit ca un compilator algebric $\mathcal{C}_{MC}:L_{ctl}\rightarrow L_M$ cu perechea de morfisme (T_{MC},H_{MC}). Morfismul $T_{MC}:Sin_{ctl}\rightarrow Sin_M$ mapează formulele CTL din algebra de cuvinte Sin_{ctl} la expresii de mulţimi din Sin_M care se evaluează la mulţimi de stări în care sunt satisfăcute formulele CTL. Morfismul $H_{MC}:Sem_{ctl}\rightarrow Sem_M$ mapează identic mulţimi de stări din Sem_{ctl} cu mulţimi de stări corespondente din Sem_M şi astfel construcţia sa este simplă.

Definiţie 3.2. [Wyk98] Orice secvenţă de variabile $x_1\in X_{c_1}$, $x_2\in X_{c_2}$,...,$x_n\in X_{c_n}$ şi orice cuvânt $cuv\in Cuv_c$ din Cuv_Σ(X) care include $x_1,x_2,...,x_n$ ca subcuvinte va defini o operaţie $d_A(cuv)$ cu signatura $c_1\times c_2\times...\times c_n\rightarrow c$ într-o algebră arbitrară A_Σ cu schema operator Σ. O asemenea operaţie $d_A(cuv)$ este numită *operaţie derivată* şi este definită pentru o asignare dată $g:\{x_1,x_2,...,x_n\}\rightarrow A$ prin homomorfismul unic $h:Cuv_{\Sigma}(\{x_1,x_2,...x_n\})\rightarrow A_{\Sigma}$ care extinde g şi astfel, evaluarea unei operaţii de derivare în A_{Σ} se realizează prin procesul definit mai sus pentru evaluarea unui homomorfism $h:Cuv_{\Sigma}(X)\rightarrow A_{\Sigma}$.

Morfismele T_{MC}: $Sin_{ctl} \rightarrow Sin_M$ şi H_{MC}: $Sem_{ctl} \rightarrow Sem_M$ astfel construite sunt reprezentate în figura 3.1 [Wyk00, RWH02, Wyk98] unde diagrama este comutativă. Această comutativitate asigură că translatarea T_{MC} păstrează semnificaţia formulelor din Sin_{ctl} atunci când acestora le sunt asociate expresii de mulţimi din Sin_M.

Maparea de pe diagonală $D_{ctl}:Sem_{ctl}\rightarrow Sin_M$ este definită prin operaţii derivate şi implementează practic translatarea formulelor CTL (din algebra de cuvinte Sin_{ctl}) la expresii de mulţimi (din Sin_M) care se evaluează la mulţimile de satisfacere ale formulelor respective în cadrul unui model dat M.

$$
\begin{tikzcd}
Sem_{ctl} \arrow[r, "\mathcal{L}_{ctl}"] \arrow[d, "H_{MC}"'] \arrow[dr, "D_{ctl}"] & Sin_{ctl} \arrow[r, "\mathcal{E}_{ctl}"] \arrow[d, "T_{MC}"] & Sem_{ctl} \arrow[d, "H_{MC}"] \arrow[dl, "D_{ctl}"'] \\
Sem_M & Sin_M \arrow[l, "\mathcal{E}_M"'] & Sem_M \arrow[l, "\mathcal{L}_M"']
\end{tikzcd}
$$

Figura 3.1. Structura algebrică a verificatorului de modele CTL $\mathcal{C}_{MC}$: $L_{ctl} \rightarrow L_M$

Construcţia homomorfismului T_{MC} poate fi realizată în cadrul unui algoritm care constituie implementarea unui verificator de modele CTL. Acest algoritm este universal deoarece este generat automat din specificarea $\langle\Sigma_{ctl}, D_{ctl}\rangle$ şi astfel metodologia descrisă poate fi aplicată pentru generarea de verificatoare de modele asociate şi altor tipuri de logici temporale.

Notăm mulţimea operatorilor din Σ_{ctl} cu $O_{ctl}^{sin} = \{\top, \perp, \neg, \wedge, \vee, \rightarrow, AX, EX, AU, EU, AG, AF, EG, EF\}$ şi respectiv $O_{ctl}^{sem} = \{S, \emptyset, C, \cap, \cup, S_I, S_{AX}, S_{EX}, S_{AU}, S_{EU}, S_{AG}, S_{AF}, S_{EG}, S_{EF}\}$.

Specificarea $\langle\Sigma_{ctl}, D_{ctl}\rangle$ se realizează prin asocierea fiecărei operaţii $o^{sin} \in O_{ctl}^{sin}$ a algebrei Sin_{ctl} cu o operaţie derivată $d_{MC}(o^{sin}) \in D_{ctl}$ din algebra Sin_M.

În continuare este descris procedeul de implementare al homomorfismului T_{MC} :

- $T_{MC}(\top) = S$, $T_{MC}(\perp) = \emptyset$, $T_{MC}(ap) = P'(ap)$ $\forall$ $ap\in AP$.
- $\forall f \in Sin_{ctl}$ astfel că $f = o^{sin}(f_1, ..., f_n)$, $T_{MC}(f) = d_{MC}(o^{sin})(T_{MC}(f_1), \ldots, T_{MC}(f_n))$.
- Deoarece între mulţimile de operatori O_{ctl}^{sin} şi O_{ctl}^{sem} există o corespondenţă biunivocă, operaţia derivată $d_{MC}(o^{sin})$ se defineşte astfel:

$$d_{MC}(o^{sin})(T_{MC}(f_1), \ldots, T_{MC}(f_n)) = d(o^{sem})(d_{ctl}(\mathcal{E}_{ctl}(f_1)), \ldots, d_{ctl}(\mathcal{E}_{ctl}(f_n))),$$

unde $o^{sem} \in O_{ctl}^{sem}$ reprezintă operaţia corespunzătoare operaţiei $o^{sin} \in O_{ctl}^{sin}$ iar $d(),d_{ctl}()$ sunt operaţii derivate.

Prin urmare, pentru definirea operaţiilor derivate $d_{MC}(o^{sin})$ se vor utiliza expresiile de mulţimi (corespunzătoare operaţiilor derivate asociate operatorilor din O_{ctl}^{sem}), expresii în care argumentele se vor reprezenta prin *meta-variabile*, astfel: pentru operaţia $o^{sin} \in O_{ctl}^{sin}$ cu signatura $\sigma(o^{sin})=c_1\times\ldots\times c_n\rightarrow$

c, operaţia derivată $d_{MC}(o^{sin})$ are ca şi parametri formali *meta-variabilele* $\$set_i$, $i=\overline{1,n}$, unde $\$set_i$ reprezintă expresia de mulţimi asociată argumentului de index *i* al operaţiei $d_{MC}(o^{sin})$. Vom utiliza *meta-variabila* $\$set$ pentru a nota expresia de mulţimi rezultată, şi astfel obţinem egalitatea:

$$\$set = d_{MC}(o^{sin})(\$set_1, \ldots, \$set_n).$$

Algoritmul de implementare al homomorfismului T_{MC}: $Sin_{ctl} \rightarrow Sin_M$ devine o particularizare a procedurii de calcul a homomorfismului generalizat, în condiţiile în care generatorii algebrei Sin_{ctl} sunt $\{\top, \bot\} \cup \{ap \mid ap \in AP\}$ şi $Car_{ctl}=\{\mathcal{F}\}$.

Procedura de calcul a homomorfismului T_{MC} necesită două funcţii suplimentare:

1. $\mathcal{A}_{\pounds} : Sin_{\mathcal{F}} \rightarrow \{\mathcal{F}\} \cup \{NULL\}$ definită astfel:

$$\mathcal{A}_{\pounds}(f) = \begin{cases} \mathcal{F}, & \textit{dacă } f \textit{ este un generator al algebrei } Sin_{ctl} \\ NULL, & \textit{altfel} \end{cases}$$

2. $\mathcal{A}_S : Sin_{\mathcal{F}} \setminus AP \rightarrow \bigcup_{n=0}^{\infty} O_{ctl}^{sin} \times Sin_{\mathcal{F}}^1 \times \ldots \times Sin_{\mathcal{F}}^n$ definită astfel:

   dacă $f = o^{sin}(f_1,\ldots,f_n)$ atunci $\mathcal{A}_S(f) = (o^{sin}, (f_1,\ldots,f_n))$

În cadrul compilatorului algebric $\mathcal{C}_{MC}: L_{ctl} \rightarrow L_M$ care reprezintă verificatorul de modele CTL, implementarea homomorfismului T_{MC}: $Sin_{ctl} \rightarrow Sin_M$ (prin care se realizează practic verificarea formulelor CTL) se poate descrie prin următoarea funcţie recursivă:

```
function TMC (f ∈ Sin_F) ∈ Sin_M {
    c := A_£ (f);
    if (c = F) {
        if ( f = ⊤)  return S;
        else if (f = ⊥) return Ø;
        else return P'(f);
    }
    else if (A_S(f)= (op, (f1,…,fn)))
        return  d_MC(op) (TMC (f1), ... , TMC(fn));
}
```

Pentru formula *f*, funcţia $\mathcal{A}_{\pounds}$ determină dacă aceasta face parte dintre generatorii algebrei Sin_{ctl}. Dacă $f \in \{\top, \bot\} \cup \{ap \mid ap \in AP\}$, $\mathcal{A}_{\pounds}(f)$ returnează $\mathcal{F}$, altfel funcţia returnează constanta *NULL*. $\mathcal{A}_S$ reprezintă un mecanism care determină operaţia şi subformulele care au fost utilizate pentru a crea formula *f*.

Componentele $\mathcal{A}_\mathcal{L}$ şi $\mathcal{A}_S$ ale compilatorului algebric $\mathcal{C}_{MC}$ pot fi implementate de un analizor lexical, respectiv de către un analizor sintactic.

Analizorul lexical $\mathcal{A}_\mathcal{L}$ trebuie să identifice atomii lexicali reprezentaţi de propoziţii atomice corect construite, conform unei gramatici regulate care generează limbajul de specificare a propoziţiilor atomice.

Analizorul sintactic $\mathcal{A}_S$ stabileşte dacă formula prezentată ca intrare verificatorului de modele *CTL* este corect construită şi aparţine limbajului formulelor *CTL* a cărui sintaxă se descrie folosind formalismul unei gramatici independente de context. Analizorul $\mathcal{A}_S$ construieşte arborele de derivare al formulei în gramatica respectivă şi astfel poate determina pentru orice subformulă a formulei date care este operaţia şi subformulele utilizate în construcţia sa.

Modul concret în care analizorul $\mathcal{A}_S$ este implicat în evaluarea T_{MC} (*f*) va fi descris în capitolul 4. Pentru implementarea componentelor $\mathcal{A}_\mathcal{L}$ şi respectiv $\mathcal{A}_S$ ale compilatorului algebric, am utilizat generatorul de analizoare ANTLR (***An****other* ***T****ool for* ***L****anguage* ***R****ecognition*).

ANTLR primeşte ca intrare o gramatică independentă de context – o descriere precisă a limbajului formulelor CTL – şi generează codul sursă al celor două analizoare într-un limbaj care poate fi specificat în fişierul de descriere a gramaticii. ANTLR suportă notaţia EBNF (Extended BNF) în cadrul meta-limbajului de descriere a gramaticilor.

Operaţiile derivate $d_{MC}()$ nu pot fi implementate ca simple operaţii derivate definite de către cuvinte ale algebrei Sin_M. În [Wyk98] se propune implementarea operaţiilor derivate prin utilizarea unui limbaj SEL (*Semantics Expression Language*) care la nevoie poate fi extins cu funcţionalităţi specifice prin intermediul sistemului TICS.

Pentru implementarea proprie a verificatorului de modele CTL, am exploatat resursele tehnologice puse la dispoziţie de sistemul ANTLR, care permite scrierea operaţiilor derivate în limbajul nativ în care este generat întreg compilatorul algebric (Java, C#, etc).

ANTLR permite specificarea sintaxei şi semanticii în cadrul aceluiaşi fişier de definire a gramaticii, permiţând ataşarea uneia sau mai multor acţiuni semantice (operaţii derivate) fiecărei reguli de producţie. De asemenea, ANTLR rezolvă automat cerinţele formulate la punctul 3. de mai sus.

Astfel, o specificare concisă a compilatorului algebric $\mathcal{C}_{MC}$ este dată de mulţimea $\{\langle r, d_{MC}(op(r))\rangle | r \in P_G\}$, unde $d_{MC}(op(r))$ este operaţia derivată corespunzătoare producţiei *r* iar *op(r)* este operatorul CTL pentru care a fost definită producţia *r*. În terminologia ANTLR, pentru $d_{MC}(op(r))$ vom folosi termenul *„acţiunea semantică ataşată producţiei r”*.

Fie $G=(N,T_G,P_G,S_0)$ o gramatică independentă de context, unde *N* este mulţimea simbolurilor neterminale, T_G – mulţimea simbolurilor terminale, P_G – mulţimea producţiilor iar S_0 – simbolul de start al gramaticii. O producţie a acestei gramatici, exprimată în notaţie BNF, este în general de forma:

$$X_0 = t_0 X_1 t_1 X_2 \ldots X_n t_n$$

unde $X_i \in N, i=\overline{0,n}$ sunt simboluri neterminale, iar $t_i \in (T_G)^*, i=\overline{0,n}$ sunt secvenţe (eventual vide) de simboluri terminale. Considerând L_G limbajul independent de context generat de către gramatica *G*, vom defini Σ_G – limbajul $L_{\Sigma G} = (Sem_{\Sigma G}, Sin_{\Sigma G}, \mathcal{L}: Sem_{\Sigma G} \rightarrow Sin_{\Sigma G})$ pe baza regulilor de specificare ale limbajului L_G.

Conform definiţiei algebrei de cuvinte Sin_{ctl}, oricărui operator care nu este nular îi corespunde o operaţie de forma $op_1^{ctl} : Sin_\mathcal{F} \times Sin_\mathcal{F} \rightarrow Sin_\mathcal{F}$ sau $op_2^{ctl} : Sin_\mathcal{F} \rightarrow Sin_\mathcal{F}$ şi are asociată o regulă de producţie de forma r_1: $X_0 = t_0 X_1 t_1 X_2 t_2$ respectiv r_2: $X_0 = t_0 X_1 t_1$ în gramatica $G=(N,T_G,P_G,S_0)$ care generează limbajul formulelor CTL. Cum operaţia asociată acestei producţii este $[op(r_1)] : [X_1] \times [X_2] \to [X_0] = op_1^{ctl}$ respectiv $[op(r_2)] : [X_1] \to [X_0] = op_2^{ctl}$, deducem că neterminalele gramaticii G au aceleaşi domenii sintactice ($Sin_{\mathcal{F}}$) şi putem considera $N=\{\mathcal{F}\}$.

În această gramatică, regulile BNF care definesc sintaxa limbajului formulelor CTL (şi implicit algebra de cuvinte Sin_{ctl}) sunt ambigue, ceea ce implică extinderea mulţimii de simboluri $Car_{ctl}=\{\mathcal{F}\}$ şi deci a mulţimii de neterminale ale gramaticii G în scopul eliminării nedeterminismului din procesul de analiză sintactică.

În teză la capitolul 4 sunt enumerate mai multe tehnici de eliminare a nedeterminismului, iar forma propusă pentru regulile EBNF ale gramaticii CTL (în sintaxă ANTLR) este:

```
ctlFormula
  :     implExpr 'au' implExpr
  |     implExpr 'eu' implExpr
  |     'ax' implExpr
  |     'ex' implExpr
  |     'af' implExpr
  |     'ef' implExpr
  |     'ag' implExpr
  |     'eg' implExpr
  |     implExpr ;

implExpr
  :     orExpr
        ( '=>' orExpr )* ;

orExpr
  :     andExpr
        ( 'or' andExpr )* ;

andExpr
  :     notExpr
        ( 'and' notExpr )* ;

notExpr
  :     'not'  atomExp
  |     atomExp ;

atomExp
  :     '(' ctlFormula ')'
  |     AP
  |     'true'
  |     'false' ;
```

Mulţimea neterminalelor gramaticii este N = {`ctlFormula`, `implExpr`, `orExpr`, `andExpr`, `notExpr`, `atomExp`}, mulţimea terminalelor este T_G = {`true`, `false`, `AP`} iar S_0 = `ctlFormula`.

Se observă utilizarea sintaxei EBNF în producţiile de rescriere ale neterminalelor `implExpr`, `orExpr` şi `andExpr`, pentru specificarea repetiţiilor (utilizarea în cadrul unei producţii a construcţiei `(item)*` specifică faptul că structura `item` se poate repeta de 0 sau mai multe ori, iar construcţia `(item)+` indică faptul că structura `item` se poate repeta de una sau mai multe ori).

Considerăm că în algebra Sin_{ctl} operatorii $\wedge$, $\vee$ respectiv $\rightarrow$ sunt *supraîncărcaţi*. Întrucât discuţia este asemănătoare pentru fiecare dintre cei trei operatori, ne vom referi în continuare doar la operatorul $\vee$.

Supraîncărcarea operatorului $\vee$ se scrie formal astfel: $\{\vee \mid \sigma_{ctl}(\vee) = \mathtt{andExpr}^n \rightarrow \mathtt{orExpr},\ n \geq 2\} \subseteq O_{ctl}$.

Regula de producţie pentru retranscrierea neterminalului `orExpr` din gramatica ANTLR de mai sus se poate echivala cu:

r_1: `orExpr = andExpr (`$\vee$` andExpr)+`
r_2: `orExpr = andExpr`

Pentru regula r_1, mulţimea operaţiilor asociate este $\{[op(r_1)] : [\mathtt{andExpr}]^n \rightarrow [\mathtt{orExpr}] \mid n \geq 2\}$.
Operatorului $\vee$ cu signatura $\sigma_{ctl}(\vee) = \mathtt{andExpr}^k \rightarrow \mathtt{orExpr}$ i se va asocia operaţia derivată $d^k_{MC}(\vee)$ a cărei evaluare $\$set := d^k_{MC}(\vee)$ $(\$set_1, \ldots, \$set_k)$ se realizează simplu:

$$\$set := \bigcup_{i=1}^{k} \$set_i$$

Regula r_2 nu implică decât o atribuire $\$set := \$set_1$, unde $\$set$ este o valoare din domeniul semantic [[`orExpr`]], iar $\$set_1$ aparţine domeniului semantic [[`andExpr`]].

În [Wyk98], pentru implementarea acţiunilor semantice se utilizează un limbaj propriu, dezvoltat ca extensie a limbajului standard macro utilizat în sistemul TICS. Acest limbaj conţine operaţii specifice din algebra ţintă (operaţii cu mulţimi) şi este implementat pornind de la o specificaţie algebrică a sa, cu instrumentele TICS.

În verificatorul nostru de modele CTL, specificarea operaţiilor semantice s-a realizat direct în limbajul Java, fără utilizarea unui limbaj intermediar. În capitolul 4 vom discuta pe larg această abordare.

Spre deosebire de abordările clasice, care utilizează analizoare bottom-up pentru limbajul formulelor CTL, în soluţia noastră (bazată pe ANTLR), analizorul sintactic $\mathcal{A}_S$ menţionat este de tip top-down. Astfel, gramatica utilizată în specificarea algebrică a limbajului L_{ctl} şi totodată în generarea limbajului independent de context L_{ctl} a fost proiectată conform cerinţelor unei analize de tip LL(*). Soluţia pentru eliminarea recursivităţii de stânga a constat în specificarea producţiilor gramaticii în sintaxă EBNF şi considerarea operatorilor $\wedge$, $\vee$ respectiv $\rightarrow$ ca fiind *supraîncărcaţi*.

Specificarea completă a compilatorului algebric a fost realizată prin prezentarea detaliată a tuturor acţiunilor semantice ataşate regulilor de producţie ale gramaticii limbajului CTL, cu o sintaxă asemănătoare meta-limbajului de definire a gramaticilor ANTLR. Acţiunile semantice, la fel ca şi operaţiile derivate din algebra Sin_M a limbajului ţintă, utilizează meta-variabile pentru evaluarea subformulelor CTL.

În cadrul acestui capitol algoritmul de verificare a fost descris mai întâi teoretic şi apoi aplicat practic pe un studiu de caz.

Utilizând instrumentul nostru de verificare a modelelor CTL, dezvoltat conform metodologiei algebrice prezentate în cadrul acestui capitol, am verificat îndeplinirea tuturor specificaţiilor formulate pentru modelul CTL aferent *excluderii mutuale a două procese*.

4. Verificarea modelelor CTL prin gramatici atributive

Procesul de verificare a unui model CTL impune definirea unei specificaţii care se reprezintă printr-o formulă CTL, urmând a se stabili dacă specificaţia respectivă este sau nu satisfăcută în cadrul modelului. O astfel de specificaţie este realizată utilizând limbajul formulelor CTL, care are la bază reguli sintactice bine stabilite. Verificarea unei formule CTL implică o translatare a acesteia, de la limbajul în care a fost definită, la limbajul peste mulţimea stărilor modelului. Rezultatul acestei translatări va consta în mulţimea de stări care satisfac formula dată în cadrul modelului CTL verificat.

De cele mai multe ori proiectarea unui translator este greu de realizat necesitând timp şi eforturi deosebite pentru construirea şi întreţinerea acestuia. Există în prezent instrumente specializate care generează întreg codul necesar pe baza unei gramatici de specificare a limbajului sursă. În mod tradiţional, instrumentele folosite pentru cele două faze ale translatării textului, *analiza lexicală* şi *analiza sintactică*, au fost LEX (*A Lexical Analyzer Generator*) şi YACC (***Y**et **A**nother **C**ompiler **C**ompiler*), sau echivalentul acestora GNU, FLEX, BISON, BYACC/J. Dezavantajul instrumentelor LEX şi YACC, respectiv FLEX şi BISON este că generează numai cod *C* şi acest cod nu este întotdeauna uşor de înţeles de către utilizator (complexitate indusă şi de natura analizoarelor pe care le generează). BYACC/J este capabil să genereze cod Java, dar acţiunile semantice suportate sunt rudimentare.

Un generator de analizoare extrem de performant este ANTLR (***An**other **T**ool for **L**anguage **R**ecognition*), capabil să genereze cod C++, C#, Java, Python şi prezentat în continuare în acest capitol. Dintre variantele enumerate vom folosi limbajul Java, reprezentând limbajul ţintă în care se va dezvolta instrumentul propriu de verificare a modelor CTL.

Câteva dintre argumentele care recomandă utilizarea gramaticilor atributive ANTLR în implementarea de verificatoare de modele sunt:

- Modelul verificat poate fi reprezentat şi accesat prin clase de obiecte în limbajul ţintă ales (C++, Java, C#, Objective C, Python), direct în specificarea gramaticii atributive.
- Pentru implementarea acţiunilor semantice în ANTLR se poate apela la întreaga putere a limbajului de programare specificat (Java, C#, etc).
- Gradul de abstractizare adus de o specificare într-un limbaj macro este substituit uşor de facilităţile limbajelor moderne şi nu este relevantă în cazul concret al algoritmilor de verificare a modelor. Astfel, nu este nevoie de crearea unor extensii ale limbajului de specificare, întrucât limbajele moderne au clase predefinite pentru operaţiile cu mulţimi.
- Se pot specifica limbaje ţintă multiple, iar acţiunile semantice pot fi implementate prin cod eficient, ţinând cont de particularităţile limbajului ales.
- Dacă se doreşte construirea unui limbaj propriu de specificare a operaţiilor derivate, *StringTemplate* reprezintă o soluţie extrem de flexibilă şi de puternică, ce emulează eficient facilităţile unui macroprocesor.

Contribuţia originală a acestei abordări constă în dezvoltarea dispozitivului de verificare a modelelor CTL prin proiectarea unei gramatici atributive ANTLR, capabilă atât să genereze limbajul de specificare a formulelor CTL cât şi să implementeze compilatorul algebric descris în capitolul precedent, definit de homomorfismul generalizat între algebra formulelor CTL şi algebra mulţimilor stărilor modelului, împreună cu operaţiile derivate corespunzătoare.

Definiţie 4.1. O *gramatică atributivă* este o gramatică independentă de context augmentată cu atribute şi reguli semantice.

Fie $G=(N,T_G,P_G,S_0)$ o gramatică independentă de context, unde N este mulţimea simbolurilor neterminale, T_G – mulţimea simbolurilor terminale, P_G – mulţimea producţiilor, iar S_0 – simbolul de start al gramaticii.

Vom nota prin $G_A=(N,T_G,P_G,S_0,A,as)$ o gramatică atributivă construită pe baza gramaticii G prin augmentarea cu atribute (A) şi reguli (acţiuni) semantice (as). O producţie $p\in P_G$ este de forma: $X_0 \to X_1 X_2 ... X_{n_p}$ unde $n_p \geq 1$, $X_0 \in N$ şi $X_k \in N \cup T_G$ pentru $1 \leq k \leq n_p$. Arborele de derivare al unei secvenţe din limbajul generat de gramatică are următoarele proprietăţi:

- Fiecare nod frunză este etichetat cu un simbol terminal din mulţimea T_G;
- Fiecare nod interior t corespunde unei producţii $p\in P_G$, iar dacă producţia este de forma $X_0 \to X_1 X_2 ... X_{n_p}$ cu semnificaţia simbolurilor descrisă mai sus, atunci t este etichetat cu simbolul X_0 şi are n_p noduri fiu etichetate cu $X_1, X_2, ..., X_{n_p}$ de la stânga la dreapta.

Gramatica atributivă utilizată în implementarea verificatorului de modele CTL va fi notată cu G_A^{CTL} şi are următoarele particularităţi:

- $A(X) = S(X), \forall\, X \in N$ – toate atributele sunt atribute de sinteză;
- $| A(X) |= 1, \forall\, X \in N$ – orice simbol neterminal X are un singur atribut, notat cu $a(X)$.
- $|as(j)|=1$, pentru fiecare $j\in \{1, \ldots, n\}$ – fiecare producţie are ataşată o singură acţiune semantică, $as(j) = \{action_j()\}$
- Considerăm că modelul CTL este dat sub forma unei structuri Kripke de forma: $M=(S, Rel, P{:}S\to 2^{AP})$, iar gramatica atributivă CTL corespunzătoare modelului M este: $G_{A,M}^{CTL}=(N, T_G,P_G,S_0,A,as)$. Atunci $AP \subseteq T_G$ (propoziţiile atomice sunt terminale ale gramaticii atributive). Fiecărei propoziţii atomice $ap \in AP$ i se asociază un singur atribut notat prin $a(ap)$, astfel că $| A(ap) |= 1\; \forall\, ap \in AP \subseteq T_G$. Mulţimea atributelor gramaticii CTL se extinde astfel:
$$A = (\bigcup_{X\in N} A(X)) \cup (\bigcup_{ap\in T_G} A(ap)) = (\bigcup_{x\in N} \{a(X)\}) \cup (\bigcup_{ap\in T_G} \{a(ap)\})$$
De asemenea, fiecărei propoziţii atomice $ap \in AP \subseteq T_G$ i se asociază acţiunea semantică $action_{ap}()$ cu rolul de a calcula valoarea atributului $a(ap)$. Mulţimea acţiunilor semantice ale gramaticii atributive CTL devine:
$$as = (\bigcup_{1\leq j\leq n} \{action_j()\}) \cup (\bigcup_{ap\in T_G} \{action_{ap}()\})$$
- pentru orice simbol $x \in N \cup AP \cup \{true, false\}$, notăm cu $v(x)$ valoarea atributului $a(x)$ şi avem $v(x) \subseteq S$ (valoarea atributului respectiv este o submulţime a mulţimii stărilor modelului M).

- Evaluarea atributelor simbolurilor neterminale este dependentă de context: dacă pentru rescrierea simbolului $X \in N$ s-a utilizat producţia i_X, atunci valoarea atributului neterminalului X în contextul respectiv este:

$$v(X) = action_{i_X}()$$

- Evaluarea simbolurilor terminale este independentă de context:

$$v(ap) = action_{ap}() = \{s \in S \mid ap \in P(s)\}\ \forall\ ap \in AP \subseteq T_G,\ v(true) = S \text{ şi } v(false) = \varnothing.$$

Deoarece adesea modelul *M* este implicit, vom scrie G_A^{CTL} în loc de $G_{A,M}^{CTL}$.

O formulă CTL *f* este corectă din punct de vedere sintactic dacă şi numai dacă există derivarea:

$$ctlFormula \overset{*}{\Rightarrow} f$$

unde am presupus *ctlFormula* = S_0 (simbolul de start al gramaticii G_A^{CTL}). Arborele de derivare corespunzător formulei *f* are frontiera compusă din simbolurile terminale care compun formula *f*.

Evaluarea unei formule CTL se realizează practic prin parcurgerea arborelui de analiză în manieră bottom-up, pornind de la frunze şi ajungând în cele din urmă la rădăcina arborelui. Rezultatul final al evaluării coincide cu valoarea atributului nodului rădăcină şi reprezintă mulţimea stărilor modelului verificat care satisfac formula dată.

În cadrul acestui capitol am demonstrat corectitudinea definirii operaţiilor derivate din algebra de sintaxă Sin_M a limbajului ţintă (asociate operatorilor CTL temporali) ca soluţii reprezentând *puncte fixe* ale unor ecuaţii.

Presupunând că formula CTL *f* este corectă sintactic, există derivarea: $ctlFormula \overset{*}{\Rightarrow} f$ utilizând producţiile gramaticii atributive G_A^{CTL}, unde *ctlFormula* este simbolul de start al acestei gramatici. Verificarea formulei *f* implică calcularea valorii atributului *a*(*ctlFormula*), notată prin $v(ctlFormula)$. Pentru o formulă dată *f*, rolul acţiunilor semantice ale gramaticii atributive G_A^{CTL} este acela de a calcula *denotaţia* lui *f* deoarece:

$$v(ctlFormula) = [\![f]\!]$$

Notăm prin $[\![f]\!]_M = \{s \in S \mid (M, s) \vDash f\}$ mulţimea tuturor stărilor din *S* care satisfac formula *f* (mulţimea stărilor în care formula *f* este adevărată). $[\![f]\!]_M$ se numeşte *denotaţia* lui *f* în modelul *M*. Deoarece adesea structura *M* este implicită, vom scrie $[\![f]\!]$ în loc de $[\![f]\!]_M$. Astfel $(M, s) \vDash f \Leftrightarrow s \in [\![f]\!]$.

Calculul denotaţiilor formulelor CTL prin determinarea punctelor fixe ale anumitor ecuaţii, după metoda descrisă în continuare, reprezintă în cele din urmă modalitatea de implementare a acţiunilor semantice ale gramaticii atributive G_A^{CTL}.

Definiţie 4.2. (Punct fix) Fie $g : 2^S \rightarrow 2^S$ şi $Z \subseteq S$ o submulţime a lui S.

1. Z se numeşte punct fix al funcţiei g dacă $g(Z) = Z$.
2. Z este cel mai mic punct fix (*LFP*) al lui g dacă $g(Z) = Z$ şi $\forall\ U \subseteq S$, $g(U) = U \Rightarrow Z \subseteq U$.
3. Z este cel mai mare punct fix (*GFP*) al funcţiei g dacă $g(Z) = Z$ şi $\forall\ U \subseteq S, g(U) = U \Rightarrow U \subseteq Z$.

Teorema de punct fix Kleene [Win93] poate fi scrisă în următoarea formă:

Teoremă 4.1. Fie $g : 2^S \rightarrow 2^S$ o funcţie monotonă definită pe o mulţime finită *S*.

1. Există un cel mai mic şi respectiv un cel mai mare punct fix al lui g.
2. $\bigcup_{n\geq 1} g^n(\phi)$ este cel mai mic punct fix al lui g.
3. $\bigcap_{n\geq 1} g^n(S)$ este cel mai mare punct fix al lui g.

Funcţia *pre-image* universală şi respectiv existenţială $pre_\forall,\ pre_\exists : 2^S \rightarrow 2^S$ este definită prin:

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

Pentru o formulă CTL *f*, verificatorul de model va calcula $[\![f]\!]$ recursiv, utilizând regulile descrise în tabelul următor, unde *LFP* şi *GFP* reprezintă cel mai mic, respectiv cel mai mare punct fix al funcţiilor specificate [SBS13]:

Formula *f*	Calculul denotaţiei $[\![f]\!]$
ap	$\{ s \in S \mid ap \in P(s) \}$
true (false)	$S\ (\phi)$
$\neg f_1$	$S \setminus [\![f_1]\!]$
$f_1 \wedge f_2$ $f_1 \vee f_2$	$[\![f_1]\!] \cap [\![f_2]\!]$ $[\![f_1]\!] \cup [\![f_2]\!]$
$AX\ f_1$	$\mathrm{pre}_\forall([\![f_1]\!])$
$EX\ f_1$	$pre_\exists([\![f_1]\!])$
$f_1 AU\ f_2$	*LFP* al $g(X) = [\![f_2]\!] \cup ([\![f_1]\!] \cap pre_\forall(X))$
$f_1 EU\ f_2$	*LFP* al $g(X) = [\![f_2]\!] \cup ([\![f_1]\!] \cap pre_\exists(X))$
$AG\ f_1$	*GFP* al $g(X) = [\![f_1]\!] \cap pre_\forall(X)$
$EG\ f_1$	*GFP* al $g(X) = [\![f_1]\!] \cap pre_\exists(X)$
$AF\ f_1$	*LFP* al $g(X) = [\![f_1]\!] \cup pre_\forall(X)$
$EF\ f_1$	*LFP* al $g(X) = [\![f_1]\!] \cup pre_\exists(X)$

De asemenea, am analizat complexitatea algoritmului de verificare a modelelor CTL. Considerăm că modelul CTL verificat *M* este definit ca un graf orientat *M* = (*S, Rel, P*) unde mulţimea de stări *S* are *V* elemente (|*S*|=*V*), iar mulţimea *Rel* se compune din *E* arce orientate (|*Rel*|=*E*). Lungimea unei formule CTL *f* constă în numărul de operatori şi operanzi care apar în componenţa acesteia şi va fi notată în continuare prin | *f* |.

Algoritmul care determină dacă starea *s*, în modelul *M*, satisface formula *f* ($(M,s) \vDash f$) necesită un timp în $O(|f| \cdot V \cdot (V+E))$.

Implementarea dispozitivului de verificare de modele CTL a fost realizată prin proiectarea unei gramatici atributive ANTLR care este capabilă să genereze limbajul de specificare a formulelor CTL [CS08, CS11]. Generarea compilatorului algebric s-a realizat prin asocierea producţiilor gramaticii cu acţiuni semantice. Construirea arborelui de derivare al unei formule CTL se realizează top-down, proces care implică activarea automată a acţiunilor semantice în scopul evaluării simbolurilor care etichetează nodurile arborelui de derivare. Evaluarea unei formule CTL se realizează practic prin parcurgerea arborelui de analiză în manieră bottom-up, pornind de la frunze şi ajungând în cele din urmă la rădăcina arborelui. Rezultatul final al evaluării coincide cu valoarea atributului nodului rădăcină şi reprezintă mulţimea stărilor modelului verificat care satisfac formula dată.

5. Arhitectura verificatorului de modele CTL. Aplicaţii. Rezultate experimentale

Contribuţia esenţială a acestui capitol constă în implementarea compilatorului algebric ca şi componentă reutilizabilă publicată ca serviciu Web [CSS11]. Forma originală a modelului CTL este transmisă compilatorului algebric $\mathcal{C}$ generat cu ajutorul ANTLR utilizând o gramatică atributivă originală. De asemenea am tratat situaţia în care clientul verifică o formulă CTL ce conţine erori sintactice. Dacă se întâmplă ca utilizatorul să solicite verificarea unei formule CTL greşită din punct de vedere sintactic, atunci clientul este notificat asupra existenţei erorii sintactice în formula transmisă serviciului Web [CSS11]. Rezultatul returnat de compilator pentru o formulă validă este mulţimea de noduri în care formula este satisfăcută.

Enumerăm în continuare câteva avantaje ale dezvoltării verificatorului de modele în arhitectură client/server bazată pe servicii Web:

- Aplicaţia client permite dezvoltarea de modele CTL în mod interactiv, într-o interfaţă grafică evoluată, dar în acelaşi timp intuitivă, simplu de utilizat;
- Crearea modelelor CTL de mari dimensiuni se poate realiza programatic, prin interfeţe API disponibile în limbajele Java şi C#;
- Pentru verificarea unui model, utilizatorul poate accesa unul dintre serviciile Web online disponibile (https://mcheck-useit.rhcloud.com/CTL-Checker/CTLCheckeService?wsdl sau http://use-it.ro/CTL-Checker/CTLCheckerService?wsdl), fără a fi necesară configurarea unui server local;
- Dacă se optează pentru configurarea locală a componentei server, instalarea acesteia este simplă şi rapidă (informaţii disponibile la http://use-it.ro/, secţiunea *Instrucţiuni de instalare*).

Versiunea curentă a instrumentului de verificare a modelor CTL se bazează pe structuri de date eficiente utilizate în reprezentarea internă a modelelor, furnizate de către biblioteca GraphStream [GrStr12].

În consecinţă, sistemul nostru permite o dezvoltare intuitivă, rapidă, interactivă sau programatică a modelelor CTL prin componenta client precum şi verificarea proprietăţilor acestor modele, specificate sub forma unor formule CTL, prin accesarea compilatorului algebric încapsulat în componenta server, utilizând tehnologia serviciilor Web.

Pentru a face disponibilă propria implementare a compilatorului algebric ca şi componentă reutilizabilă a instrumentului de verificare a modelelor CTL, o vom publica cu ajutorul unui serviciu Web.

Arhitectura serviciului Web implementat este descrisă în figura 5.1 [CSS11].

Serviciul Web va recepţiona de la client reprezentarea XML a modelului CTL împreună cu o formulă CTL *f*.

Forma originală a modelului CTL (graf orientat) este apoi transmisă compilatorului algebric $\mathcal{C}$, generat cu ajutorul ANTLR [Parr07], utilizând o gramatică atributivă originală CTL [CSS11]. Rezultatul returnat de compilator este $\mathcal{C}(f) = \{s \in S| \ (M,s) \models f\}$, şi reprezintă mulţimea de noduri în care formula este satisfăcută.

Evident, formula f poate conţine erori sintactice. Întrucât dorim notificarea clientului asupra existenţei unei posibile erori sintactice în formula transmisă serviciului Web, trebuie să suprascriem comportamentul implicit de tratare a erorilor ANTLR.

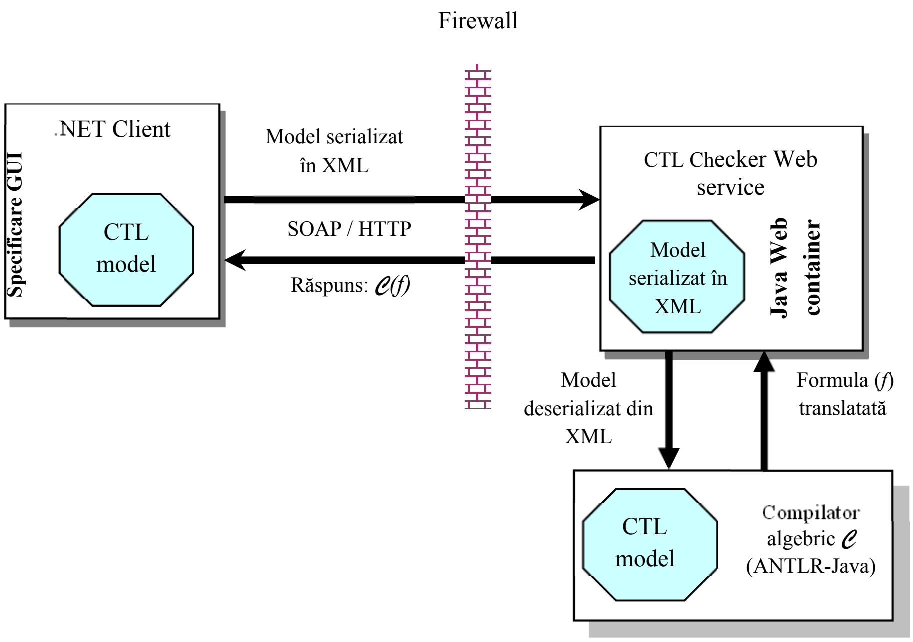

Figura 5.1. Arhitectura serviciului Web care furnizează verificatorul de modele

Verificatorul de modele CTL se bazează pe un client cu interfaţă grafică [CSS11], dezvoltat în C#, care permite specificarea grafică, interactivă, a modelelor CTL.

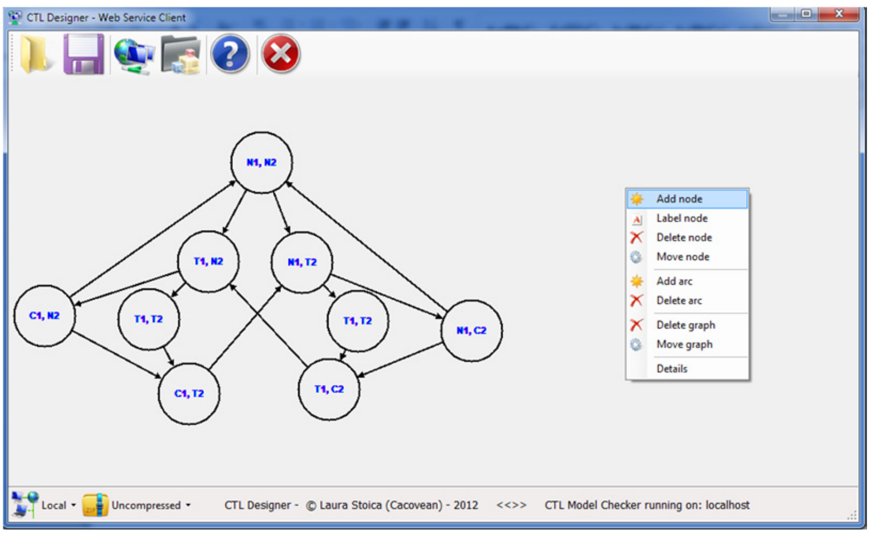

Figura 5.2. CTL Designer – client implementat în C#

În plus, CTL Designer permite realizarea anumitor configurări prin controale ale propriei interfeţe:

Icon	Semnificaţie
	Permite selectarea serverului Web: local sau la distanţă (Internet). Detalii despre localizarea şi accesarea serviciilor Web disponibile online în Internet pot fi găsite la: http://use-it.ro
	Permite selectarea serverului Internet implicit.
	În cazul modelelor de mari dimensiuni, este recomandată activarea compresiei acestora înainte de a fi trimise serviciului Web.

Modelul CTL este transmis ca şi document XML serviciului Web, împreună cu formula care trebuie verificată.

Arhitectura generală a sistemului implementat este descrisă cu ajutorul unei diagrame de pachete în limbajul UML:

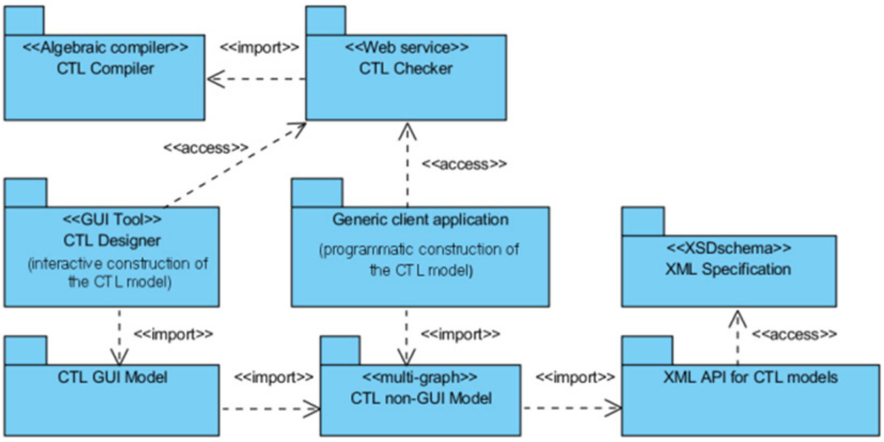

Figura 5.3. Arhitectura de sistem a verificatorului de modele CTL

Instrumentul de verificare a modelelor CTL conţine următoarele pachete:

- Compilatorul CTL încapsulat în cadrul serviciului Web (*CTL Checker*); implementarea atât a compilatorului cât şi a serviciului Web este realizată în limbajul Java.
- Clientul cu interfaţă grafică implementat în C#, utilizat pentru construcţia interactivă a modelelor CTL ca şi grafe orientate (*CTL Designer*).
- În cazul modelelor CTL de mari dimensiuni, cu multe stări, este necesară o construcţie programatică a acestora. Pachetul *CTL non-GUI model* conţine clasele utilizate pentru reprezentarea internă a unui model CTL ca şi graf orientat. Sunt disponibile două interfeţe de programare (API), pentru C# şi respectiv pentru Java. Pentru reprezentarea internă a unui model CTL în C#, implementarea noastră este bazată pe structuri de date descrise în [Ebe87], mai exact pe liste de adiacenţă memorate simetric pentru parcurgere înainte şi înapoi. Implementarea Java este bazată pe GraphStream [GrStr12].
- Pachetul *XML API for CTL* conţine clasele necesare codificării modelelor CTL în XML.

- Pachetul *CTL GUI Model* este responsabil cu reprezentarea grafică a structurilor Kripke ca şi grafe orientate.

Am proiectat un algoritm pentru determinarea unei strategiei câştigătoare pentru jocul *X0* jucat de către doi oponenţi, printr-o modelare adecvată a jocului şi utilizarea verificatorului de modele CTL pentru a determina mulţimile de satisfacere ale unor formule CTL care formalizează câştigarea / necâştigarea jocului. Pentru ca un jucător să urmeze strategia câştigătoare, prin mutarea aleasă trebuie să determine o tranziţie într-o stare aflată în mulţimea de satisfacere a formulei verificate.

Algoritmul pentru determinarea strategiei optime

Presupunem că jocul este în starea $s_0 \in S$ şi notăm cu k numărul poziţiilor libere de pe tabla de joc. Strategia jucătorului *X* poate fi exprimată prin algoritmul următor:

Pas 1	Determină toate stările din model care satisfac formula: $(\mathrm{AX\ EX})^{k/2}(\mathrm{AX}\ \overline{111})$, pentru a alege mutarea care favorizează câştigarea jocului în viitor. Vom nota această mulţime cu WIN1.
Pas 2	Determină toate stările din model care satisfac formula: EX $\overline{222}$, pentru a evita ca jucătorul *O* să câştige la mutarea următoare. Vom nota această mulţime cu WIN2.
Pas 3	Dacă $(\mathrm{WIN1} \setminus \mathrm{WIN2}) \cap \mathrm{succ}(s_0) \neq \emptyset$ atunci Alege aleator o stare s din mulţimea rezultat. altfel dacă $\mathrm{succ}(s_0) \setminus \mathrm{WIN2} \neq \emptyset$ atunci Alege aleator o stare s din mulţimea rezultat. altfel Alege aleator o stare s din mulţimea $\mathrm{succ}(s_0)$. sfd Setează s ca şi stare curentă ($s_0 := s$).
Step 4	Dacă $\overline{111} \in P(s)$ atunci STOP. Jucătorul *X* a câştigat. Dacă tabla este plină, atunci este declarată egalitate şi STOP.
Step 5	Jucătorul *O* efectuează o mutare. Dacă $\overline{222} \in P(s)$ atunci STOP. Jucătorul *O* a câştigat. Dacă tabla *este* plină, atunci este declarată egalitate şi STOP, altfel salt la pasul 1.

Aplicaţia implementată (accesibilă din clientul *CTL Designer*) reprezintă o bună oportunitate de a studia eficienţa abordărilor noastre în proiectarea şi implementarea unui verificator de modele CTL. Rezultatele experimentale care permit evaluarea performanţelor verificatorului nostru de modele au fost prezentate în secţiunea 5.4.

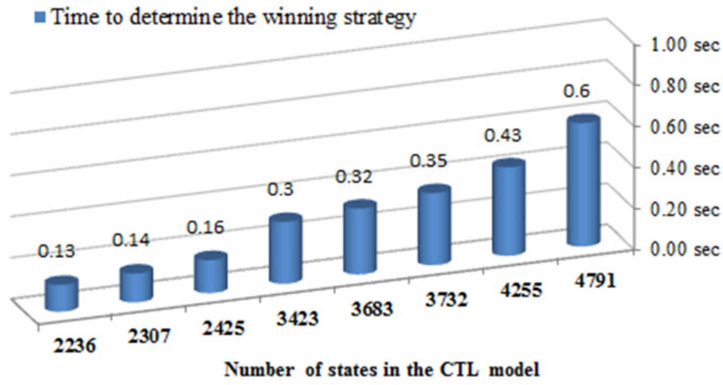

Figura 5.4. Evaluarea performanţei verificatorului de modele CTL

6. Dispozitiv de verificare a modelelor ATL folosind descrierea algebrică

Dispozitivul de verificare a unui model este o tehnologie utilizată frecvent pentru verificarea automată a sistemelor. Algoritmii de verificare a modelelor sunt în prezent implementaţi în diferite medii sau limbaje de programare. Sistemul verificat poate fi un sistem fizic sau un program concurent în timp real.

Comportamentul unui sistem închis poate fi descris prin modelul Kripke. O structură Kripke oferă un model natural pentru calculele dintr-un *sistem închis*, al cărui comportament este complet determinat de starea sistemului. Modelarea compoziţională şi proiectarea unor sisteme reactive cer ca fiecare componentă să fie privită ca un *sistem deschis*.

Definiţie 6.1. Un *sistem deschis* este un sistem care interacţionează cu mediul său şi al cărui comportament depinde de starea sistemului, precum şi de comportamentul mediului.

În scopul de a construi modele potrivite pentru sisteme deschise, s-a definit logica ATL [AHK02].

ATL extinde CTL prin înlocuirea cuantificatorilor de drum $\forall$ şi $\exists$ prin modalităţile de cooperare $\langle\langle\mathcal{A}\rangle\rangle$, unde $\mathcal{A}$ este o mulţime de agenţi. O formulă $\langle\langle\mathcal{A}\rangle\rangle\ \varphi$ exprimă faptul că mulţimea de agenţi $\mathcal{A}$ are o strategie colectivă pentru a pune în aplicare φ.

ATL este o logică temporală cu timp ramificat, care descrie în mod natural procesările efectuate de sistemele multi-agent şi execuţia jocurilor cu mai mulţi jucători (în cazul nostru îi vom numi agenţi). Logica ATL oferă o cuantificare selectivă cu privire la drumurile (căile de execuţie ale) programului care sunt rezultate posibile ale jocurilor [AHK02]. ATL foloseşte formule cu timp alternativ pentru a construi verificatoare de modele în scopul de a aborda probleme cum ar fi receptivitatea, realizabilitatea şi controlul.

Complexitatea verificatorului de model ATL, peste structurile fără constrângeri de corectitudine, este liniară în ceea ce priveşte mărimea structurii jocului şi lungimea formulei [AHK02].

Având un limbaj ATL bine definit, implementarea dispozitivului de verificare de modele ATL va fi echivalată cu un compilator algebric care translatează o formulă a modelului ATL la o mulţime de noduri, peste care formula ATL este satisfăcută [SS11, SSS12].

Structura Kripke reprezintă un model natural pentru sistemele închise, independent de descrierea abstractă, de nivel înalt, a acestor sisteme (dată de exemplu ca o compunere de maşini cu stări).

Prin analogie, modelul natural pentru compoziţiile sistemelor deschise este *structura jocurilor concurente* [AHK02, KP05]. Astfel, spre deosebire de CTL care este interpretat peste structurile Kripke, ATL este interpretat peste structurile jocurilor concurente. În scopul de a captura compoziţii de sisteme deschise, considerăm jocuri cu mai mulţi jucători în care mulţimea de jucători reprezintă o componentă diferită a sistemului şi a mediului.

Metodologia algebrică de proiectare a unui verificator de modele CTL, prezentată în detaliu în capitolul 3, a fost aplicată cu succes pentru proiectarea unui verificator de modele ATL ca un compilator algebric $\mathcal{C}_{\mathcal{MC}}:L_{atl}\rightarrow L_{Set}$ unde limbajul sursă L_{atl} este limbajul formulelor ATL, iar limbajul ţintă L_{Set} este limbajul care descrie mulţimea de stări a modelului ATL verificat (reprezentat printr-o structură de joc concurent) în care formulele respective sunt satisfăcute [SS11,SB12,SSS12].

Specificarea completă a compilatorului algebric a fost realizată prin prezentarea detaliată a tuturor acţiunilor semantice ataşate regulilor de producţie ale gramaticii limbajului ATL, cu o sintaxă asemănătoare meta-limbajului de definire a gramaticilor ANTLR. Acţiunile semantice, la fel ca şi operaţiile derivate din algebra Sin_{Set} a limbajului ţintă, utilizează meta-variabile pentru evaluarea subformulelor ATL. Consideraţiile prezentate în capitolul 4 privind utilizarea unei gramatici atributive ANTLR pentru generarea verificatorului de modele CTL rămân valabile şi pentru logica ATL.

Pentru funcţia Pre($\mathcal{A}$, ρ), care apare în toate operaţiile derivate asociate operatorilor ATL modali, am realizat o formalizare folosind concepte ale Algebrei Relaţionale (RA). Aceasta a permis translatarea expresiilor obţinute în declaraţii SQL care pot fi executate pe un server de baze de date performant, exploatând astfel un optimizator foarte eficient al interogărilor SQL. Evaluarea performanţelor obţinute a fost realizată în capitolul 7.

Pentru o structură de joc concurent $\mathcal{S}$ se poate defini un multi-graf orientat $G_S = (X,U)$, unde $X=Q$, şi $(b,e) \in U \Leftrightarrow \exists\langle j_1,\ldots,j_k\rangle \in \mathcal{D}(b)$ astfel încât $\delta(b,j_1,\ldots,j_k) = e$. Funcţia de etichetare pentru un graf G_S este definită după cum urmează: $\mathcal{L}:U\rightarrow\overline{\mathcal{M}}$, $\forall$ $u=(b,e) \in U$, $\mathcal{L}(u) = \langle j_1,\ldots,j_k\rangle$ unde $\delta(b,j_1,\ldots,j_k) = e$.

Definim schema de relaţie $(B{:}Q_B, M_1{:}D_1, \ldots, M_k{:}D_k, E{:}Q_E)$ unde $Q_B = \{b \in Q \mid \exists e \in Q$ astfel încât $(b, e) \in U\}$, $Q_E = \{e \in Q \mid \exists b \in Q$ astfel încât $(b, e) \in U\}$ şi D_i, $i \in \{1, \ldots, k\} = \Lambda$ au fost definite în ecuaţia 6.2.1, astfel că dacă R_S este un nume de relaţie cu schema definită mai înainte, $(B{:}b, M_1{:}j_1, \ldots, M_k{:}j_k, E{:}e) \in R_S \Leftrightarrow \langle j_1,\ldots,j_k\rangle=\mathcal{L}((b,e))$.

Pentru o mulţime $\mathcal{A}$ de m agenţi, $\mathcal{A}\subseteq\Lambda$, $\mathcal{A} = \{i_1, \ldots, i_m\}$, definim:

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

$$R_{\mathcal{L}}(\mathcal{A})= \pi_{B,LABEL\leftarrow M_{i_1}\circ\ldots\circ M_{i_m},E}(R_S(\mathcal{A}))$$

unde operatorul ° poate fi definit astfel: i ° j = i || ',' || j.

Pentru o mulţime $\Theta \subseteq Q_E$, $b \in$ Pre($\mathcal{A}$, Θ) $\Leftrightarrow\exists\, j_{i_l} \in d_{i_l}(b)$, $i_l\in \mathcal{A}$, $l=\overline{1,m}$ şi $\exists e \in\Theta$ astfel încât

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

şi $\nexists$ e'$\in Q_E \setminus \Theta$ astfel încât

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

Cu alte cuvinte, b$\in$Pre($\mathcal{A}$, Θ) $\Leftrightarrow \exists\, j_{i_l} \in d_{i_l}(b)$, $i_l \in \mathcal{A}$, $l=\overline{1,m}$ astfel încât

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

În continuare, mulţimea de stări $Q_E \setminus \Theta$ este notată cu $\bar{\Theta}$.

Putem acum să proiectăm un algoritm de calcul pentru funcţia Pre($\mathcal{A}$, Θ) folosind expresii RA:

Pasul 1:	$\pi_{B,\,LABEL}(\sigma_{E\in\Theta}(R_{\mathcal{L}}(\mathcal{A}))) = R_{\mathcal{L}}^{\Theta}(\mathcal{A})$ $\pi_{B,\,LABEL}(\sigma_{E\in\bar{\Theta}}(R_{\mathcal{L}}(\mathcal{A}))) = R_{\mathcal{L}}^{\bar{\Theta}}(\mathcal{A})$
Pasul 2:	$\rho_x(R_{\mathcal{L}}^{\Theta}(\mathcal{A})) \underset{x.B\,=\,y.B\ \wedge\ x.LABEL\,=\,y.LABEL}{⟖} \rho_y(R_{\mathcal{L}}^{\bar{\Theta}}(\mathcal{A})) = R_{\mathcal{L}}^{\Theta,\bar{\Theta}}(\mathcal{A})$
Pasul 3:	$\sigma_{y.LABEL=null}(\pi_{x.B,\,y.LABEL}\, R_{\mathcal{L}}^{\Theta,\bar{\Theta}}(\mathcal{A})) = R_{\mathcal{L}}^{\Theta,null}(\mathcal{A})$
Pasul 4:	$\mathrm{Pre}(\mathcal{A}, \Theta) = \pi_{x.B}(R_{\mathcal{L}}^{\Theta,null}(\mathcal{A}))$

Figura 6.1. Calcularea funcţiei Pre($\mathcal{A}$, Θ) folosind expresii ale algebrei relaţionale

Algoritmul prezentat în figura 6.1 poate fi implementat în limbajul SQL astfel:

```
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
```

Figura 6.2. Calcularea funcţiei Pre($\mathcal{A}$, Θ) folosind declaraţii SQL

Implementarea algoritmului de verificare a modelelor ATL este bazat pe codul Java generat de ANTLR folosind o gramatică atributivă originală [SB12,SSS12] şi care asigură tratarea eventualelor erori lexicale/sintactice apărute în formula ce trebuie analizată.

Utilizarea ANTLR pentru generarea compilatorului algebric a permis implementarea tuturor operaţiilor derivate în cod Java, iar pentru operaţiile specifice mulţimilor au fost utilizate clase Java din pachete predefinite. De asemenea, modelul ATL poate fi accesat direct din codul corespunzător operaţiilor derivate. Flexibilitatea soluţiei bazată pe utilizarea limbajului Java ca limbaj ţintă în care este generat compilatorul algebric permite accesarea standardizată, prin drivere JDBC, a unei game largi de servere de gestiune a bazelor de date (SGBD-uri) care pot fi astfel utilizate de algoritmul de verificare a modelelor ATL. În prezent, verificatorul nostru de modele ATL suportă MySQL, SQL Server şi H2 ca servere de date.

În secţiunea 6.1 este prezentat un model ATL original pentru problema secţiunii critice, care se bazează pe un mutex gestionat de un proces supervizor (eventual sistemul de operare). Soluţia noastră îmbunătăţeşte modelul CTL clasic deoarece suportă adevărata concurenţă: două procese pot solicita simultan intrarea în secţiunea critică, iar accesul este restricţionat prin intermediul mecanismului de sincronizare amintit.

6.1. Model ATL pentru problema secţiunii critice rezolvată folosind un mutex

Considerăm modelul nostru prezentat în figura 6.1.1 ca şi structură de joc concurent $\mathcal{S}=\langle k,Q,\Pi,\pi,\mathcal{M},d_\Lambda,\delta\rangle$, vom detalia semantica pentru simbolurile din Π - mulţimea de propoziţii (etichetele din nodurile ce reprezintă stările) şi $\mathcal{M}$ – mulţimea mutărilor agenţilor. Avem $\Pi = \{I_1, I_2, W_1, W_2, E_1, E_2, L_1, L_2, F\}$ cu următoarele semnificaţii:

- I_i – procesul i este în starea **I**dle (*inactiv*), $i = \overline{1,2}$;
- W_i – procesul i este în starea **W**ait (*aşteaptă* să intre în secţiunea critică), $i = \overline{1,2}$;
- E_i – procesul i este în starea **E**xecute (*execută* codul din secţiunea critică), $i = \overline{1,2}$;
- L_i – mutex-ul este deţinut (**L**ock) (*blocat*) de procesul i, $i = \overline{1,2}$;
- F – mutex-ul nu este deţinut de nici un proces (**F**ree) (*liber*).

Simbolurile din mulţimea $\mathcal{M} = \{l, e, i, f\} \cup \{pd, dp, p\text{-}, \text{-}p\}$ au următoarea semnificaţie:

- l – solicită intrarea în secţiunea critică (**l**ock) (*blochează mutexul*);
- e – solicită să **e**xecute codul din secţiunea critică;
- i – nu există solicitări pentru secţiunea critică (**i**dle);
- f – eliberarea mutex-ului (**f**ree), părăseşte secţiunea critică;
- pd – (**p**ermission) permisiune pentru primul agent, (**d**eny)refuz pentru al doilea agent;
- dp – (**p**ermission) permisiune pentru al doilea agent, (**d**eny) refuz pentru primul agent;
- p- – (**p**ermission) permisiune pentru primul agent, agentul al doilea este inactiv (fără cerere);
  -p – (**p**ermission) permisiune pentru al doilea agent, primul agent este inactiv (fără cerere);

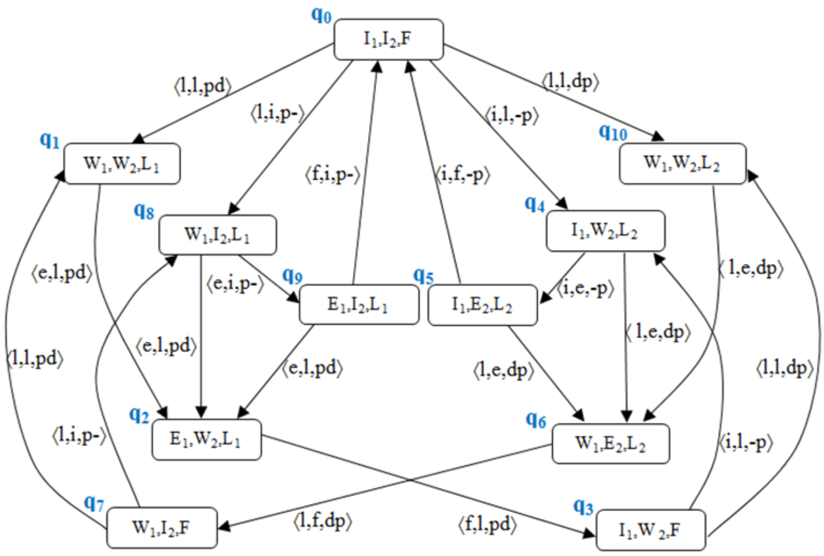

Figura 6.1.1. Modelul ATL pentru cele două procese concurente care doresc să intre în secţiunea critică

Folosind instrumental nostru de verificare de modele am demonstrat că următoarele formule ATL sunt satisfăcute de modelul prezentat mai sus.

(i) **Siguranţa** – Procesele nu rulează simultan instrucţiuni din secţiunea critică.

$$\text{not}(\langle\langle \mathcal{A} \rangle\rangle \sim (E_1 \text{ and } E_2))$$

(ii) **Garanţia** – de fiecare dată când un proces încearcă să intre în secţiunea critică (deţinând mutex-ul), i se garantează accesul în viitor.

$$W_i \Rightarrow \text{not } (\langle\langle \mathcal{A} \rangle\rangle \# (\text{not } E_i)),\ i = \overline{1,2}$$

(iii) **Neblocarea** – fiecare proces poate solicita oricând intrarea în secţiunea critică.

$$\text{not } (\langle\langle \mathcal{A} \rangle\rangle \sim (\text{not } (I_i \Rightarrow \langle\langle \mathcal{A} \rangle\rangle @\ W_i))),\ i = \overline{1,2}$$

(iv) **Fără succesiuni impuse** – procesele nu au restricţii ca să intre alternant în secţiunea critică.

$$\langle\langle \mathcal{A} \rangle\rangle \sim (E_1 \text{ and } (\langle\langle \mathcal{A} \rangle\rangle\ E_1 \text{ U } (\text{not } E_1 \text{ and } (\langle\langle \mathcal{A} \rangle\rangle \text{ not } E_2 \text{ U } E_1))))$$

$$\langle\langle \mathcal{A} \rangle\rangle \sim (E_2 \text{ and } (\langle\langle \mathcal{A} \rangle\rangle\ E_2 \text{ U } (\text{not } E_2 \text{ and } (\langle\langle \mathcal{A} \rangle\rangle \text{ not } E_1 \text{ U } E_2))))$$

(v) **Deţinerea mutex-ului** – Un proces poate să execute secţiunea critică doar dacă deţine mutex-ul.

$$E_i \Rightarrow L_i,\ i = \overline{1,2}$$

(vi) **Eliberarea mutex-ului** – Dacă unul dintre procese deţine mutex-ul, în viitor procesul trebuie să îl elibereze.

$$\text{not}(\langle\langle \mathcal{A} \rangle\rangle \sim (\text{not } ((L_1 \text{ or } L_2) \Rightarrow \text{not } (\langle\langle \mathcal{A} \rangle\rangle \# (\text{not } F)))))$$

(vii) **Concurenţa** – Dacă nu avem nici un proces în secţiunea critică, ambele procese pot solicita simultan intrarea în secţiunea critică, fără a se bloca unul pe celălalt.

$$I_1 \text{ and } I_2 \Rightarrow \langle\langle \mathcal{A} \rangle\rangle @\ (W_1 \text{ and } W_2)$$

În continuare vom aplica algoritmul din figura 6.1 pentru calcularea funcţiei *Pre()* cu argumente diferite, transmise în procesul de verificare a două formule ATL prezentate anterior.

Exemplu 6.1.1. Pentru modelul ATL prezentat în figura 6.1.1 verificăm formula ATL notată cu (ii):

$$W1 \Rightarrow \text{not } (\langle\langle \mathcal{A} \rangle\rangle \# \text{ (not } E1))$$

cu semnificaţia prezentată în (ii). Algoritmul de verificare a modelului va solicita câteva apeluri ale funcţiei *Pre()* cu anumite argumente. În tabelul 6.1.1 sunt prezentate două computaţii (calcule) ale funcţiei *Pre()*:

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

Tabel 6.1.1. Calculul funcţiei Pre($\mathcal{A}$, Θ) la verificarea formulei ATL (ii)

Pentru $\mathcal{A}$ = {1} deoarece $i \in d_1(0)$, $\pi_E(B{:}0,\ M_1{:}i,\ E{:}Q_E) = \{(E{:}4)\}$, şi $4 \in \Theta \Rightarrow 0 \in$ Pre($\mathcal{A}$, Θ).

Pentru $\mathcal{A}$ = {2}, $d_2(0)$ = {l, i}. Avem $\pi_E(B{:}0,\ M_2{:}i,\ E{:}Q_E) = \{(E{:}8)\}$, dar $8 \notin \Theta$. De asemenea, $\pi_E(B{:}0,\ M_2{:}l,\ E{:}Q_E) = \{(E{:}1), (E{:}4), (E{:}10)\}$ dar $1 \notin \Theta$. Concludem că $0 \notin$ Pre($\mathcal{A}$, Θ).

O contribuţie teoretică originală în abordarea verificatoarelor de modele ATL o reprezintă utilizarea conceptelor de algebre relaţionale în cadrul compilatorului algebric, pentru implementarea operaţiilor derivate asociate operatorilor ATL modali. Traducerea expresiilor algebrice respective în SQL a fost naturală, iar rezultatul obţinut s-a integrat perfect în arhitectura client/server bazată pe servicii Web a noului sistem implementat şi descris pe larg în capitolul 7.

7. Arhitectura verificatorului de modele ATL. Aplicaţii. Evaluarea performanţelor

În cadrul acestui capitol am publicat componenta de bază a verificatorului de modele ATL, compilatorul algebric, în cadrul unui serviciu Web pentru a furniza un sistem complet, bazat pe tehnologii robuste (Java, .NET, SQL) şi pe standarde bine cunoscute (XML, XSD, SOAP, HTTP).

Serviciul Web va recepţiona de la client reprezentarea XML a unui model ATL împreună cu o formulă ATL dată φ. Rezultatul furnizat de către serviciul Web (în cazul în care formula φ este corectă din punct de vedere sintactic), în urma invocării compilatorului algebric $\mathcal{C}$, va fi $\mathcal{C}(\varphi) = \{q \in Q \mid q \models \varphi\}$

– mulţimea stărilor modelului dat în care formula φ este satisfăcută.

În figura 7.1 este prezentată diagrama de clase a implementării instrumentului de verificare a modelelor ATL. Clasele ATLParser şi ATLLexer sunt generate de către ANTLR, utilizând ca intrare gramatica atributivă a limbajului (formulelor) ATL. Rolul clasei ATXml este să decodeze reprezentarea XML a modelului ATL pentru a o trimite parser-ului, împreună cu formula ATL care trebuie evaluată. Clasa ATLChecker conţine metoda serviciului Web care este invocată de către client, având ca parametri modelul ATL, o formulă ATL şi un set $\mathcal{A}$ de agenţi.

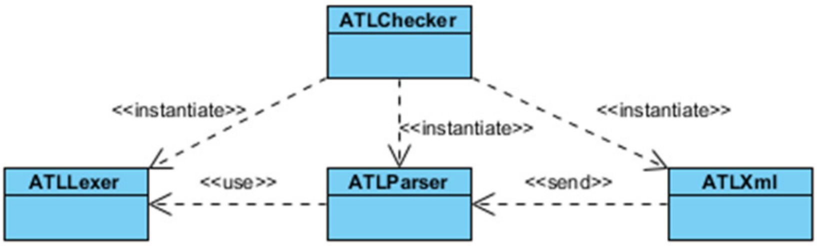

Figura 7.1. Diagrama de clase a instrumentului ATL Checker

Un alt motiv pentru dezvoltarea verificatorului de modele ATL pe partea de server este reprezentat de implementarea internă a funcţiei *Pre*() şi care se bazează exclusiv pe un server de date în vederea execuţiei.

Implementarea serviciului Web este bazată pe GlassFish ca şi container Web, şi pe MySQL/SQL Server/H2 ca server de baze de date.

Instrumentul nostru de verificare de modele este bazat pe un client C# GUI care permite dezvoltarea de modele ATL într-un mod grafic interactiv. Pentru reprezentarea internă a unui model ATL ca şi multi-graf orientat, implementarea noastră este bazată pe structuri de date descrise în [Ebe87]. Astfel, codarea unui model ATL se bazează pe liste de adiacenţă simetrice pentru înlănţuire înainte şi înapoi. Această paradigmă suportă un mod orientat pe arce de manevrare a garfurilor cu arce multiple. Arhitectura implementării serviciului Web a verificatorului de modele ATL [SB12, SSS12] este asemănătoare cu cea de la serviciul Web a verificatorului de modele CTL [CSS11].

Pentru o mai bună înţelegere a procesului de verificare a unui model ATL, în figura 7.2 este reprezentată diagrama cazurilor de utilizare a instrumentului nostru de verificare a modelelor ATL:

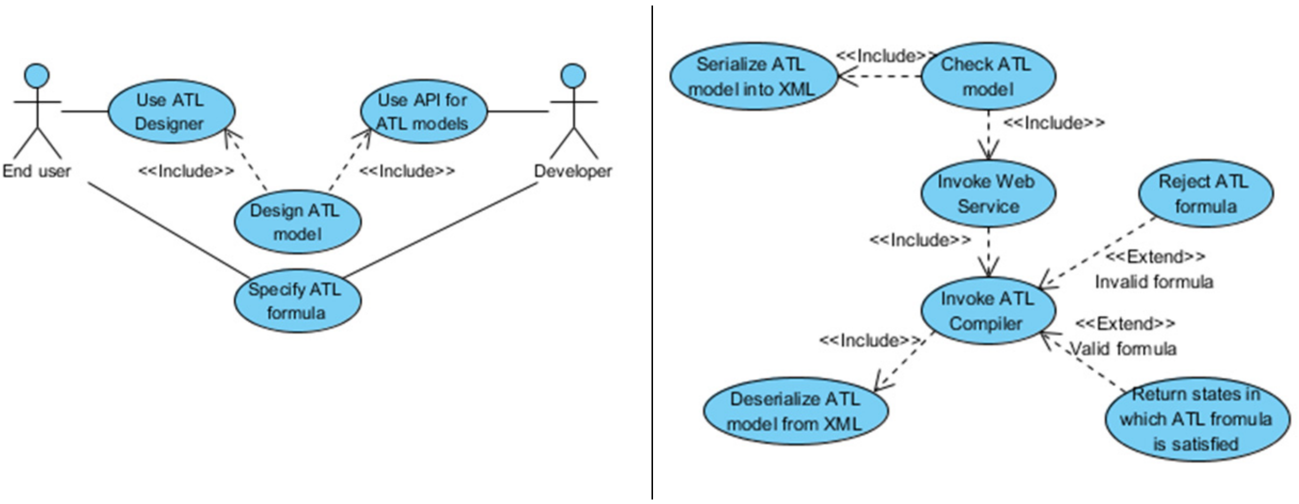

Figura 7.2. Diagrama cazurilor de utilizare a instrumentului ATL Checker

Serviciul Web va recepţiona de la client reprezentarea XML a unui model ATL împreună cu o formulă ATL. După deserializare, forma originală a modelului ATL este transmisă compilatorului algebric generat de către ANTLR utilizând gramatica noastră ATL extinsă. Pentru o formulă corectă din punct de vedere sintactic, compilatorul va returna ca şi rezultat setul de stări în care formula este satisfăcută. Dacă formula ATL nu este validă, serviciul Web va returna un mesaj care descrie eroarea depistată.

Implementarea C# a componentei client a instrumentului nostru (ATL Designer) permite o specificare grafică interactivă a modelului ATL ca şi multi-graf orientat. Componenta server a instrumentului original de verificare a modelelor (ATL Checker) a fost publicată ca serviciu Web, expunând funcţionalitatea sa clienţilor prin interfeţe XML standard [SB12, SSS12].

Pentru verificarea unui model, utilizatorul poate accesa unul dintre serviciile Web online disponibile (https://mcheck-useit.rhcloud.com/ATL-Checker/ATLCheckeService?wsdl sau http://use-it.ro/ATL-Checker/ATLCheckerService?wsdl), fără a fi necesară configurarea unui server local. Dacă se optează pentru configurarea locală a componentei server, instalarea acesteia este simplă şi rapidă (informaţii disponibile la http://use-it.ro/, secţiunea *Instrucţiuni de instalare*).

Pentru construirea de modele ATL de mari dimensiuni s-au dezvoltat interfeţe de programare obiectuale, care permit specificarea programatică a modelelor ATL în format XML, disponibile pentru limbajele C# şi Java.

În acest scop, este necesară specificarea unui document XSD (*XML Schema Definition*) – schema XML utilizată pentru definirea şi validarea structurii şi tipurilor de date ale elementelor din cadrul documentelor XML care reprezintă modele ATL recunoscute de sistemul de verificare a modelelor prezentat în capitolul 6.

Pe baza schemei XML se poate genera cod C# pentru construirea programatică a unui model ATL reprezentat sub forma unui document XML, utilizând *Microsoft Xml Schemas/DataTypes support utility* (xsd.exe). Diagrama de clase generate pe baza documentului XSD este prezentată în figura 7.3:

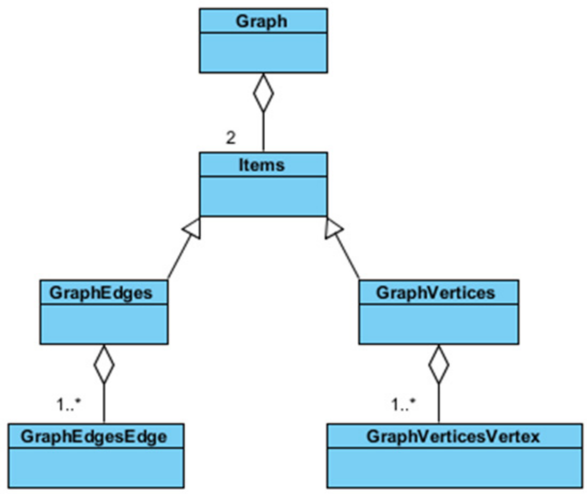

Figura 7.3. Diagrama de clase asociată codului generat de *Microsoft Xml Schemas/DataTypes support utility*

Codul obţinut în urma procesului descris mai sus a fost inclus cu mici extensii în cadrul instrumentului de verificare a modelelor, în scopul testării funcţionalităţii oferite. Stabilirea extremităţilor unui arc din clasa *GraphEdgesEdge* se fac prin setarea atributelor *from*, respectiv *to*.

Forma finală a modelului ATL generat poate fi vizualizată în ATL Designer. Toate elementele modelului care sunt relevante doar pentru modul grafic de reprezentare au fost iniţializate cu valori nule, după cum se poate observa în figura 7.4.

Procesul de verificare a unei formule ATL pentru un model generat programatic este identic cu cel asociat unui model construit în interfaţa grafică ATL-Designer.

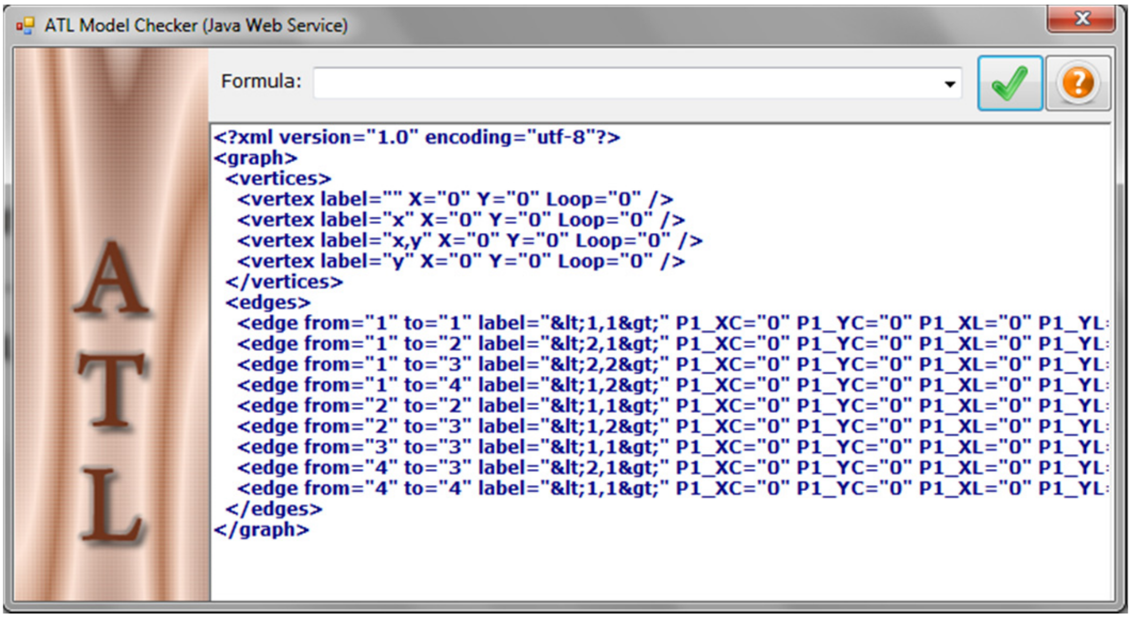

Figura 7.4. Modelul construit automat prin interfaţa API

Pentru a furniza o descriere generală a arhitecturii de sistem a instrumentului de verificare a modelelor ATL prezentat în teză, utilizăm o diagramă UML de pachete (*UML package diagram*), prezentată în figura 7.5:

Se poate observa integrarea interfeţei API pentru construirea programatică a modelelor ATL în cadrul arhitecturii generale a întregului sistem dezvoltat pentru verificarea modelelor ATL, care conţine următoarele pachete (de clase):

- Compilatorul algebric (ATL Compiler) încapsulat în cadrul serviciului Web (ATL Checker);
- Aplicaţia grafică client utilizată în construirea interactivă a modelelor ATL sub forma unor multi-grafe orientate (ATL Designer);
- În cazul modelelor ATL de mari dimensiuni, cu multe stări, este necesară construirea programatică a acestor modele. Pachetul *ATL non-GUI model* conţine clasele utilizate pentru reprezentarea internă a unui model ATL ca şi multi-graf orientat, pe baza unor liste de adiacenţă simetrice pentru accesare înainte-înapoi;
- Pachetul *XML API for ATL models* conţine clasele necesare pentru reprezentarea modelului ATL în format XML. Cea mai mare parte a codului acestui pachet a fost generate folosind aplicaţia *Microsoft Xml Schemas/Data Types support utility* (xsd.exe), căreia i-a fost furnizată ca intrare schema XSD de specificare a reprezentării XML a unui model ATL generic;

- Pachetul *ATL GUI Model* este responsabil cu reprezentarea grafică a structurilor de joc concurente reprezentate ca multi-grafe orientate (desenarea arcelor prin curbe Bézier, etc.).

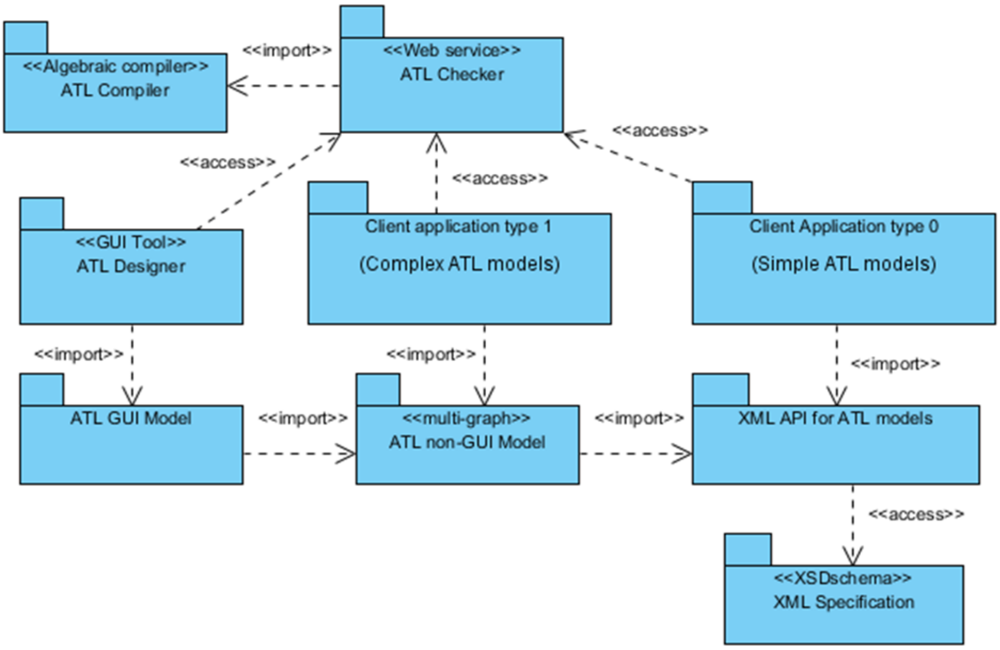

Figura 7.5. Arhitectura generală a verificatorului de modele ATL

În încheierea capitolului s-a abordat problema determinării strategiilor optime în sisteme multi-agent modelate ca structuri sincrone de jocuri concurente alternative. Tehnica prezentată poate fi aplicată cu succes şi pentru alte clase de modele ATL.

Am proiectat un algoritm pentru determinarea unei strategiei câştigătoare pentru jocul *X0* jucat de către doi oponenţi, printr-o modelare adecvată a jocului şi utilizarea verificatorului de modele ATL pentru a determina mulţimile de satisfacere ale unor formule ATL care formalizează câştigarea / necâştigarea jocului. Pentru ca un jucător să urmeze strategia câştigătoare, prin mutarea aleasă trebuie să determine o tranziţie într-o stare aflată în mulţimea de satisfacere a formulei verificate.

Algoritmul pentru determinarea strategiei optime pentru jucătorul 1 este:

Pasul 1:	Determină toate stările din cadrul modelului care satisfac formula: $\langle\langle 1\rangle\rangle\Diamond$ (111), pentru a alege mutarea care favorizează câştigul în viitor. Notăm această mulţime cu WIN1.
Pasul 2:	Determină stările din cadrul modelului care satisfac formula: $\langle\langle 2\rangle\rangle\circ$ (222), pentru a împiedica jucătorul 2 să câştige la următoarea mutare. Notăm această mulţime cu WIN2.
Pasul 3:	Dacă (WIN1\WIN2) $\cap$ succ(q) $\neq \emptyset$, atunci Alege aleatoriu o stare q din mulţimea rezultată. Setează starea curentă ca fiind starea q.
	Altfel, dacă succ(q)\WIN2 $\neq \emptyset$, atunci Alege aleatoriu o stare q din mulţimea rezultată. Setează starea curentă ca fiind starea q.
	Altfel

	Alege aleatoriu o stare q din mulţimea *succ(q)*. Setează starea curentă ca fiind starea q.
Pasul 4:	Dacă $\overline{111} \in \pi(q)$, STOP. Jucătorul 1 a câştigat. Dacă tabla este plină, se declară egalitate şi STOP, altfel după ce mută jucătorul 2 se reia pasul 1 (dacă jucătorul 2 nu a câştigat sau tabla nu s-a completat).

Performanţa verificatorului de modele ATL a fost evaluată în raport cu diverse servere de date: MySql, SQL Server sau H2.

Timpul total pentru determinarea strategiei câştigătoare în modelul ATL, Intel Core I5, 2.5 GHz, 4Gb RAM			
Nr stări	SQL Server 2008 (sec)	MySQL 5.5 (sec)	H2 1.3 (sec)
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

Tabelul 7.1. Analiza comparativă a impactului serverelor de date în performanţa verificatorului de modele ATL

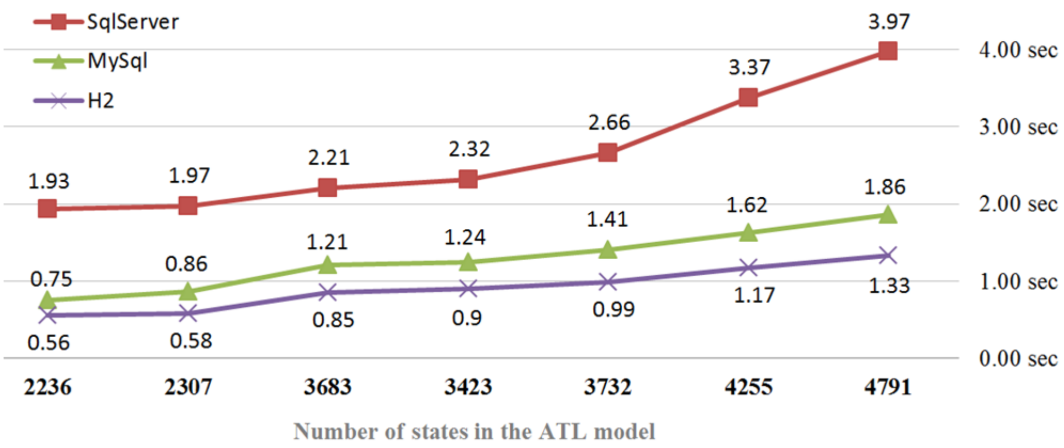

Figura 7.6. Analiza comparativă a impactului serverelor de date în performanţa verificatorului de modele ATL

În [OM03] este realizată o comparaţie între Lurch (un verificator de modele cu căutare aleatorie) şi două verificatoare de modele consacrate, SMV şi SPIN, în care se prezintă necesarul de timp şi memorie precum şi acurateţea atinsă de fiecare instrument în simularea jocului X0.

SPIN este un verificator de modele LTL (Linear Temporal Logic) cu stări explicite, iar SMV este un verificator CTL simbolic.

Deşi logicile LTL şi CTL sunt interpretate în mod natural peste structurile Kripke, care furnizează modele de calcul ale sistemelor închise iar logica ATL este utilizată pentru specificarea şi verificarea sistemelor deschise, din punct de vedere teoretic expresivitatea superioară a ATL (în cazul sistemelor închise, ATL degenerează în CTL) nu induce costuri suplimentare – complexitatea verificării modelelor ATL este liniară în raport cu dimensiunea modelului şi respectiv lungimea formulei verificate [AHK02].

Rezultatele din [OM03] arată că atât SMV cât şi SPIN au fost capabile să găsească o strategie optimală pentru unul dintre jucători în mai puţin de o secundă, pe o tablă de joc 3x3.

După cum se poate observa din tabelul 7.1, verificatorul nostru de modele ATL este comparabil cu instrumentele LTL/CTL, dar trebuie avut în considerare faptul că modelul ATL este mai expresiv (cu ATL se poate cuantifica performanţa individuală a unui jucător sau a unei echipe de jucători cooperanţi, modelele ATL surprind diverse aspecte ale interacţiunilor sincrone sau asincrone dintre sisteme deschise, etc.).

În lucrarea [Rua08], jocul *X0* a fost implementat în limbajul RML (Reactive Modules Language). RML este limbajul de descriere a modelelor ATL în verificatorul de modele MOCHA, care a fost proiectat de către Alur (şi colaboratori) [AHMQRT98]. Rezultatele experimentale au arătat că timpul necesar pentru determinarea unei strategii câştigătoare pentru un jucător a fost de 1 minut şi 6 secunde pe o confuguraţie cu un procesor Intel Dual-Core 1.8 Ghz. Rulat pe aceeaşi cofiguraţie, verificatorul nostru de modele ATL este capabil să găsească o strategie câştigătoare în aproximativ 4 secunde utilizând MySql ca şi server de bază de date şi respectiv în 2 secunde dacă a fost utilizat H2.

Utilizând o tehnologie bazată pe servere de date în cadrul componentei principale a verificatorului de modele ATL (compilatorul algebric), instrumentul nostru se bazează pe o solidă fundaţie pentru îmbunătăţiri ulterioare ale aspectelor legate de performanţă şi scalabilitate.

În stadiul actual de dezvoltare, rezultatele experimentale sunt încurajatoare, arătând faptul că înstrumentul nostru de verificare a modeleler ATL este capabil să gestioneze în mod eficient sisteme de mari dimensiuni.

În încheierea capitolului, am dezvoltat o tehnică de validare a comportamentelor de tip FSM (Finite State Machine) ale agenţilor JADE. Soluţia propusă se bazează pe versiunea Java a componentei ATL Library, care permite verificarea unor specificaţii exprimate prin formule ATL în cadrul unui model ATL construit în mod automat, odată cu definirea maşinii cu stări finite a comportamentului JADE.

Verificarea automată a unui sistem multi-agent prin modele ATL este procesul formal prin care o specificaţie dată, exprimată printr-o formulă ATL şi reprezentând o proprietate comportamentală dorită, este satisfăcută în cadrul modelului ATL al sistemului respectiv.

Pentru început, vom prezenta un model ATL adecvat unui comportament compus de tip FSM (Finite State Machine) care este înclus în cadrul unui agent JADE. Acest model permite elaborarea regulilor de mapare între conceptele ATL şi JADE. Biblioteca ATL pentru limbajul Java va fi folosită pentru a verifica corectitudinea proiectării agenţilor JADE care posedă comportamente de tip FSM, pentru a evita apariţia scenariilor incorecte ca urmare a unei proiectări defectuoase.

Definiție 7.1. O maşină cu stări finite JADE este un tuplu FSM=$(Q_{FSM}, \Gamma, \gamma, q_0, F, t, \delta_{FSM})$ unde:

- Q_{FSM} este o mulțime finită și nevidă de *stări*;

- Γ reprezintă o mulțime finită de *nume de stări*;
- γ: $Q_{FSM} \rightarrow \Gamma$ este numită funcția de ***etichetare***, definită astfel: pentru fiecare stare $q \in Q_{FSM}$, $\gamma(q) \in \Gamma$ reprezintă *numele* stării q;
- q_0 este un element ce reprezintă starea inițială din Q_{FSM};
- $F \subseteq Q_{FSM}$ este o mulțime de stări finale;
- t: $Q_{FSM} \rightarrow 2^{\mathbb{Z} \cup \{default\}}$ este funcția de ***terminare***, unde pentru fiecare stare $q \in Q_{FSM}$, $t(q) \subseteq \mathbb{Z} \cup \{default\}$ reprezintă mulțimea de coduri de terminare admisibile ale stării q;
- Funcția de tranziție $\delta_{FSM}(q, j)$, asociază la fiecare stare $q \in Q_{FSM}$ și la fiecare cod de terminare j al stării q, starea de tranziţie din starea q în cazul în care comportamentul fiu asociat cu starea q returnează la terminare valoarea j.

Comportamentul unei maşini cu stări finite este mai uşor de înţeles când aceasta este reprezentată grafic sub forma unui graf orientat (numit şi diagramă de tranziţie a stărilor). Stările sunt reprezentate prin vârfurile grafului, iar regulile de tranziţie sunt specificate prin arcele (orientate ale) grafului. Fiecare tranziţie din starea q este etichetată cu codul de terminare al stării q care declanşează tranziţia. Arcul fără o stare sursă indică starea iniţială a sistemului (starea q_0).

Pentru o mașină cu stări finite JADE structura de joc concurent echivalentă $\mathcal{S} = \langle k, Q, \Pi, \pi, \mathcal{M}, d_\Lambda, \delta \rangle$ este definită după cum urmează:

- Există doar un singur agent, prin urmare $k = 1$ şi $\Lambda = \{1\}$;
- Mulțimea de stări este $Q = Q_{FSM}$;
- Mulțimea finită de *propoziții* este definită de $\Pi = \Gamma \cup \{ *FINAL* \}$;
- Funcția de ***etichetare*** π: $Q \rightarrow 2^{\Pi}$ este definită astfel:

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

- Mulțimea finită și nevidă de mutări $\mathcal{M}$ conține toate codurile de terminare admisibile, prin urmare:

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

- Funcția de ***mutări alternative*** d_Λ: $\Lambda \times Q \rightarrow 2^{\mathcal{M}}$ este definită astfel: $d_\Lambda(1, q) = t(q) \forall q \in Q$
- Funcția de tranziție δ este definită după cum urmează:

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

În timpul unei reacţii a maşinii cu stări finite este declanşată o anumită tranziţie, aleasă din mulţimea tranziţiilor admisibile (tranziţii care pleacă din starea curentă) şi a cărei etichetă reprezintă evenimentul de terminare a stării curente. Noua stare curentă a maşinii cu stări finite va fi starea destinaţie a tranziţiei declanşate.

Dacă evenimentul de terminare a stării curente $q \notin F$ nu este asociat explicit nici unei tranziţii admisibile, atunci:

- dacă există tranziţia admisibilă etichetată cu *default*, se va declanşa această tranziţie, numită şi *tranziţie implicită*;
- altfel maşina trece într-o stare inconsistentă.

În cazul în care maşina ajunge într-o stare $q \in F$, după terminarea activităţilor din starea respectivă, execuţia maşinii se opreşte.

Deoarece testarea şi simularea pot doar să crească încrederea în implementarea unui sistem software, dar nu pot demonstra că toate erorile au fost găsite, vom utiliza o metodă formală – verificarea de modele ATL – pentru detectarea şi eliminarea erorilor în proiectarea comportamentelor de tip FSM ale unui agent JADE.

Validarea proiectării unui sistem prin utilizarea ATL presupune verificarea faptului că modelul ATL corespunzător satisface cerinţele de sistem exprimate prin formule ATL.

Pentru un comportament FSMBehaviour al unui agent JADE, verificarea modelului ATL este realizată în doi paşi:

1. În prima etapă, este construit modelul ATL pentru o mașină cu stări finite JADE
2. Apoi, se verifică dacă o specificaţie dată sub forma unei formule ATL, care reprezintă o proprietate de comportament dorită, este satisfăcută în cadrul modelului obţinut la pasul anterior.

Utilizând biblioteca ATL dedicată limbajului Java din cadrul instrumentului de verificare al modelelor ATL [CS13], se poate realiza atât construirea cât şi verificarea automată a modelului ATL pentru o specificaţie dată. Astfel, se pot detecta stările modelului în care formula nu este satisfăcută, ceea ce impune o reproiectare a maşinii cu stări finite pe care se bazează comportamentul FSMBehaviour al agentului JADE.

8. Concluzii și direcții viitoare

În prima parte a lucrării au fost analizate atât din punct de vedere teoretic cât şi din perspectiva implementării aspecte generale legate de realizarea dispozitivului de verificare de modele *CTL* folosind metodologia algebrică.

În proiectarea verificatorului de modele CTL am avut în vedere suportul acestuia pentru sintaxa completă CTL, astfel încât compilatorul algebric implementat să recunoască toţi operatorii CTL. Această facilitate conduce la o mai bună expresivitate a specificaţiilor care trebuie verificate în cadrul unui model CTL dat.

Un avantaj semnificativ oferit de metodologia algebrică în proiectarea verificatorului de modele CTL este reprezentat de faptul că toate componentele dispozitivului de verificare sunt în mod automat generate pornind de la specificarea algebrică $\langle \Sigma_{ctl}, D_{ctl} \rangle$ care cuprinde schema operator a limbajului sursă (limbajul formulelor CTL) şi respectiv operaţiile derivate asociate operatorilor CTL, considerate ca operaţii în algebra de sintaxă a limbajului ţintă.

Indicaţiile teoretice pentru implementarea dispozitivului de verificare de modele CTL reprezintă o contribuţie esenţială în cadrul acestei lucrări, acestea aducând numeroase elemente de noutate faţă de alte produse similare existente la ora actuală, cum ar fi: utilizarea unor operaţii dependende de model specifice ($pre_\forall()$ şi respectiv $pre_\exists()$), implementarea unui mecanism de observare a procesului de căutare a soluţiei, interactivitate în proiectarea şi verificarea modelelor, accesarea funcţionalităţii din aplicaţii externe folosind interfaţa de programare (API), etc. De asemenea, verificatorul de modele CTL a fost proiectat astfel încât înţelegerea, utilizarea şi extinderea acestuia să fie extrem de simple, putând fi astfel folosit cu uşurinţă în activităţi didactice sau de cercetare.

Spre deosebire de abordările clasice, care utilizează analizoare bottom-up pentru limbajul formulelor CTL, în soluţia noastră (bazată pe ANTLR), analizorul sintactic este de tip top-down. Astfel, gramatica utilizată în specificarea algebrică a limbajului formulelor CTL a fost proiectată conform cerinţelor unei analize de tip LL(*) (detalii au fost oferite în capitolul 4). De asemenea, pentru specificarea producţiilor gramaticii s-a utilizat sintaxa EBNF, iar pentru scrierea acţiunilor semantice s-a utilizat limbajul Java.

O altă contribuţie originală prezentată în cadrul acestei lucrări este reprezentată de arhitectura bazată pe servicii Web a instrumentului de verificare a modelelor CTL, a cărui implementare este descrisă pe larg în capitolul 5 şi care este bazată pe tehnologii robuste (Java, .NET) şi pe standarde cunoscute: XML, SOAP, HTTP. Astfel, compilatorul algebric CTL, încapsulat într-un serviciu WEB şi bazat pe codul Java generat de ANTLR folosind o gramatică atributivă CTL originală, realizează verificarea unei formule CTL în cadrul unui model precizat, furnizând totodată semnalarea eventualelor erori lexicale/sintactice în formula analizată.

Ca facilitate importantă a sistemului menţionăm capabilitatea de specificare grafică, interactivă, a unui model CTL, folosind clientul C# denumit *CTL Designer*, precum şi construirea programatică a modelelor CTL de mari dimensiuni prin biblioteci API disponibile în limbajele C# şi Java.

În capitolul 6 al acestei teze am descris conceptele teoretice necesare implementării unui verificator de modele ATL. ATL este o extensie CTL extrem de utilă în modelarea sistemelor deschise, care se bazează pe conceptul de structură de joc concurent.

De asemenea, am arătat cum metodologia algebrică utilizată în proiectarea unui verificator de modele CTL poate fi aplicată cu succes în dezvoltarea verificatorului de modele ATL.

O contribuţie teoretică originală în abordarea verificatoarelor de modele ATL o reprezintă utilizarea conceptelor de algebre relaţionale în cadrul compilatorului algebric, pentru implementarea operaţiilor derivate asociate operatorilor ATL modali. Traducerea expresiilor algebrice respective în SQL a fost naturală, iar rezultatul obţinut s-a integrat perfect în arhitectura client/server bazată pe servicii Web a noului sistem implementat şi descris pe larg în capitolul 7.

Testele noastre, efecuate cu baza de date H2 în cadrul verificatorului de modele ATL au confirmat performanţele crescute ale acestei abordări. Deşi până în prezent tehnologia bazelor de date nu a fost exploatată în cadrul verificatoarelor de modele, considerăm ca progresele tehnologice în domeniu pot conduce la reconsiderarea acestei opţiuni.

Verificatorul de modele ATL implementează facilităţile amintite ale verificatorului CTL şi extinde funcţionalitatea standard a componentei client (ATLDesigner) printr-o interfaţă de programare (API) care permite încorporarea sistemului de verificare a modelelor ATL în aplicaţii multi-agent, în scopul determinării de strategii optime ale agenţilor prin verificarea de formule exprimate în limbajul logicii temporale ATL, în cadrul unor modele construite dinamic.

Exploatând arhitectura bazată pe componente a verificatorului nostru de modele ATL, am dezvoltat o tehnică de validare a comportamentelor de tip FSM (Finite State Machine) ale agenţilor JADE. Soluţia propusă se bazează pe versiunea Java a componentei ATL Library, care permite verificarea specificaţiilor exprimate prin formule ATL în cadrul unui model construit în mod transparent, odată cu definirea maşinii cu stări finite a comportamentului JADE.

Ambele instrumente sunt dezvoltate în arhitectură client-server. Implementările .NET ale componentelor client permit construirea interactivă a modelelor CTL, respectiv ATL în cadrul unor interfeţe grafice. Componentele server sunt publicate ca servicii Web şi sunt dispobibile online la adresele: https://mcheck-useit.rhcloud.com, http://use-it.ro.

De asemenea, pentru construirea programatică a modelelor de mari dimensiuni, sunt disponibile interfeţe de programare API încapsulate în biblioteci disponibile pentru limbajele C# şi Java.

Ca direcţii viitoare de cercetare ne propunem:

- Crearea unor biblioteci care să includă diverse modele CTL/ATL care pot fi verificate şi studiate de către utilizator;
- Crearea unei extensii a verificatorului de modele CTL care să includă restricţii de timp;
- Dezvoltarea unui verificator de modele CTL simbolic;
- Crearea unei extensii a verificatorului de modele ATL care să includă restricţii de timp;
- Dezvoltarea unui verificator de modele ATL simbolic;
- Deşi dispozitivele de verificare de modele dezvoltate până în prezent au fost folosite pentru verificarea unor sistemele complexe, o limitare majoră a acestor abordări este că instrumentele respective pot doar verifica corectitudinea unei specificaţii a sistemului. Cu alte cuvinte, dacă sunt identificate erori de către dispozitivul de verificare a unui model în cadrul unui sistem specificat, sarcina corectării sistemului este în totalitate lăsată proiectanţilor de sistem. Prin urmare, dispozitivul de verificare este în general folosit doar pentru a verifica dacă un sistem este corect, fără însă a-l modifica dacă verificarea acestuia eşuează. Corectarea automată a unui sistem a fost abordată în lucrările [SW96, DNM06, CPPB08], pentru anumite cazuri specifice. O posibilă direcţie de cercetare o reprezintă modificarea automată a unui model în cazul în care verificarea acestuia a eşuat.
- Dezvoltarea de modele teoretice şi extensii ale logicilor temporale abordate în cadrul lucrării în vederea lărgirii domeniului de aplicare a tehnologiei de verificare a modelelor la sisteme software de mare actualitate: aplicaţii/servicii Web, servicii pentru Web-ul semantic (Semantic Web) [Mar04], sisteme de gestiune a bazelor de date distribuite/autonome pentru aplicaţii Web, etc.

Complexitatea crescândă a aplicaţiilor Web face ca acestea să fie mai vulnerabile în raport cu numărul şi gravitatea erorilor care pot să apară în timpul funcţionării, compromiţând robusteţea, corectitudinea şi încrederea acordată de către utilizatori. Se impune deci necesitatea dezvoltării de modele şi tehnici de verificare a acestora, care vizează aplicaţiile Web în general şi serviciile Web în special.

Ca o direcţie viitoare a cercetărilor noastre, ne propunem extinderea sistemelor de verificare a modelelor prezentate în această lucrare în vederea generării automate sau cel puţin asistate a specificaţiilor care trebuie verificate (formule exprimate momentan în limbajul asociat unei logici temporale), prin intermediul unei interfeţe grafice prietenoase şi interactive.

Fiecare din aceste aspecte presupune provocări complexe atât din punct de vedere teoretic cât şi practic, însă importanţa domeniului de cercetare abordat în cadrul lucrării reprezintă o motivaţie suficientă pentru a le trata prin continuarea rezultatelor obţinute până acum.

Ne manifestăm speranţa că rezultatele teoretice originale obţinute în cadrul acestei lucrări, integrate şi exploatate în cadrul implementării propriilor instrumente software pentru verificarea modelelor, pot contribui la realizarea unui deziderat mult aşteptat: integrarea tehnicilor de verificare formală în ciclul curent de proiectare şi dezvoltare a sistemelor software.

BIBLIOGRAFIE

[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.

[M∅l04] A. M∅ller. 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.