Third International Conference
Modelling and Development of Intelligent Systems
October 10 - 12, 2013
"Lucian Blaga" University
Sibiu - Romania

Using ATL model checking in agent-based applications

Laura Florentina Stoica, Florin Stoica, Florian Mircea Boian

Abstract

Verification of a model executes an exhaustive search of errors in the state-space of the model, in order to verify that this model satisfies the correctness requirements. This search can be accomplished automatically, providing the answer if the verified requirement is satisfied within the model or is appearing a violation. In this paper we present an advanced technique to verify JADE software agents, using Alternating-time Temporal Logic. The proposed solution is based on our original ATL model checker.

1 Introduction

In order to build a well-functioning software system, a thorough investigation of requirements is needed and as a result the requirement specifications are obtained. After the conceptual design phase results an abstract design specification which needs to be validated and checked against the requirement specifications. More specifically, design validation involves checking whether a system design satisfies the system requirements.

Verification of a software system involves checking whether the system follow its design specifications.

Both of these tasks, system verification and design validation can be accomplished thoroughly and reliably using model-based formal methods, such as model checking [1].

Main concern of formal methods in general, and model checking in particular, is helping to design correct systems [2]. The correctness of a software system cannot be proved only by testing. The well-known statement by Dijkstra states that:

"Testing can only show the presence of errors, never their absence".

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

Alur et al. introduced Alternating-time Temporal Logic (ATL), a more general variety of temporal logic, suitable for specifying requirements of multi-agent systems [3]. The semantics of ATL is formalized by defining games such that the satisfaction of an ATL formula corresponds to the existence of a winning strategy.

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

ATL defines "cooperation modalities", of the form <<A>> φ, where A is a group of agents. The intended interpretation of the ATL formula <<A>> φ is that the agents A can cooperate to ensure that φ holds (equivalently, that A have a winning strategy for φ) [4].

ATL has been implemented in several tools for the analysis of open systems. In [5] is presented a verification environment called MOCHA for the modular verification of heterogeneous systems.

The input language of MOCHA is a machine readable variant of reactive modules. Reactive modules provide a semantic glue that allows the formal embedding and interaction of components with different characteristics [5].

In [6] is described MCMAS, a symbolic model checker specifically tailored to agent-based specifications and scenarios. MCMAS has been used in a variety of scenarios including web-services, diagnosis, and security. MCMAS takes a dedicated programming language called ISPL (Interpreted Systems Programming Language) as model input language [6].

In [7] is presented a new interactive ATL model checker environment based on algebraic approach. The broad goal of our research was to develop a reliable, easy to maintain, scalable model checker tool to improve applicability of ATL model checking in design of general-purpose computer software.

Our ATL model checker tool is using oriented multi-graphs to represent concurrent game structures over which is interpreted the ATL specification language. The core of our ATL model checker is the ATL compiler which translates a formula *f* of a given ATL model to set of nodes over which formula *f* is satisfied. We found that our ATL model checker tool scale well, and can handle even very large problem sizes efficiently, mainly because it is based on a client/server architecture and take advantage of a high performance database server for implementation of the ATL model checker algorithm.

In this paper we will show how ATL model checking technology can be used for automated verification of multi-agent systems, developed with JADE.

The paper is organized as follows. In section 2 we present the definition of the concurrent game structure, the ATL syntax and the ATL semantics. In section 3 is presented the JADE FSMBehaviour. These concepts are applied in section 4 where ATL Library is used to verify the design of the JADE agents having FSM - driven behaviours. Conclusions are presented in section 5.

2 Alternating-Time Temporal Logic

The ATL logic was designed for specifying requirements of open systems. An open system interacts with its environment and its behaviour depends on the state of the system as well as the behaviour of the environment. In the following we will describe a computational model appropriate to describe compositions of open systems, called concurrent game structure (CGS).

2.1 The concurrent game structure

A *concurrent game structure* is defined as a tuple $S=\langle \Lambda, Q, \Gamma, \gamma, M, d, \delta \rangle$ with the following components:

- a nonempty finite set of all agents $\Lambda = \{1, \ldots, k\}$;
- Q denotes the finite set of *states* ;
- Γ denotes the finite set of *propositions* (or *observables*);
- γ: $Q \rightarrow 2^{\Gamma}$ is called the ***labelling*** (or *observation*) function, defined as follows: for each state $q \in Q$, $\gamma(q) \subseteq \Gamma$ is the set of propositions *true* at q;
- M represents a nonempty finite set of moves;
- the ***alternative moves*** function d: $\Lambda \times Q \rightarrow 2^{M}$ associates for each player $a \in \{1,\ldots,k\}$ and each state $q \in Q$ the set of available moves of agent a at state q. In the following, the set $d(a,q)$ will

be denoted by $d_a(q)$. For each state $q \in Q$, a tuple $\langle j_1,\ldots,j_k\rangle$ such that $j_a \in d_a(q)$ for each player $a \in \Lambda$, represents a *move vector* at q.

- the transition function $\delta(q,<j_1,\ldots,j_k>)$, associates to each state $q \in Q$ and each move vector $\langle j_1,\ldots,j_k\rangle$ at q the state that results from state q if every player $a \in \{1,\ldots,k\}$ chooses move j_a.

A *computation* of S is an infinite sequence $\lambda = q_0, q_1,\ldots$ such that q_{i+1} is the successor of q_i, $\forall i \geq 0$. A *q-computation* is a computation starting at state q. For a computation λ and a position $i \geq 0$, we denote by $\lambda[i]$, $\lambda[0,i]$, and $\lambda[i,\infty]$ the i-th state of λ, the finite prefix $q_0, q_1,\ldots,q_i$ of λ, and the infinite suffix $q_i, q_{i+1} \ldots$ of λ, respectively [3].

2.2 Syntax of ATL

The ATL operator $\langle\langle\ \rangle\rangle$ is a path quantifier, parameterized by sets of agents from Λ. The operators $\circ$ ('next'), $\Box$ ('always'), $\Diamond$ ('future') and U ('until') are temporal operators. A formula $\langle\langle \mathcal{A} \rangle\rangle \varphi$ expresses that the team $\mathcal{A}$ has a collective strategy to enforce φ.

The temporal logic ATL is defined with respect to a finite set of agents Λ and a finite set Γ of propositions. An ATL formula has one of the following forms:

(*1*) p, where $p \in \Gamma$;
(*2*) $\neg \varphi$ or $\varphi_1 \vee \varphi_2$ where φ, φ_1 and φ_2 are ATL formulas;
(*3*) $\langle\langle \mathcal{A} \rangle\rangle \circ \varphi$, $\langle\langle \mathcal{A} \rangle\rangle \Box \varphi$, $\langle\langle \mathcal{A} \rangle\rangle \Diamond \varphi$ or $\langle\langle \mathcal{A} \rangle\rangle \varphi_1 \, U \, \varphi_2$, where $\mathcal{A} \subseteq \Lambda$ is a set of players, and φ, φ_1 and φ_2 are ATL formulas.

Other boolean operators can be defined from $\neg$ and $\vee$ in the usual way. The ATL formula $\langle\langle \mathcal{A} \rangle\rangle \Diamond \varphi$ is equivalent with $\langle\langle \mathcal{A} \rangle\rangle$ *true* $U \varphi$.

2.3 Semantics of ATL

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

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

the set of available moves of agent a within the game structure S.

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

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

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

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

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

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

3 Automated verification of an agent-based system

Our ATL model checker tool contains the following packages:

- ATL Compiler – the core of our tool, embedded into a Web Service (ATL Web Service);
- ATL Designer – the GUI client application used for interactive construction of the ATL models as directed multi-graphs;
- ATL Library – used for development of custom applications with large ATL models. Versions of this library are provided for C# and Java.

The software can be downloaded from http://use-it.ro (binaries and examples of use):

Download			Description
ATL	**ATL Designer**		Tool for an interactive design of the ATL models
	ATL Web Service		The Web service used for checking the ATL models
	C#	**ATL Library**	The C# Class Library (Visual Studio 2010)
		Example of use	Example of use (Visual Studio 2010 Project)
		Source code	C# source code for testing the ATL Library
		Zip Library [1])	Class library for compression in zip format of ATL models
	Java	**ATL Library**	The Java Class Library (JDK version 7)
		Example of use	Example of use (NetBeans 7.2 Project)
		Source code	Java source code for testing the ATL Library
		GraphStream Library [2])	Packages used for internal representation of the ATL models

Fig. 1 Download page of the site hosting our ATL model checker tool

In the following we will show how our tool can be used for applying the ATL technology in the field of agent-based applications.

Because ATL includes notions of agents, their abilities and strategies (conditional plans) explicitly in its models, ATL is appropriate for planning, especially in multi-agent systems [8].

Automated verification of a multi-agent system by ATL model checking is the formal process through which a given specification expressed by an ATL formula and representing a desired behavioural property is verified to hold for the ATL model of that system.

In the following, ATL Library will be used to detect errors in the design, specification and implementation of an agent developed in JADE.

The ATL Library (java version) will be used to validate the design of JADE agents having FSM-behaviours, in other words, to see that no incorrect scenarios arise as a consequence of a bad design.

3.1 Jade agents with FSM behaviours

The JADE platform is a middleware that facilitates the development of multi-agent systems and applications conforming to FIPA standards for intelligent agents [10].

The Agent class represents a common base class for user defined agents. Therefore, from the programmer's point of view, a JADE agent is simply an instance of a user defined Java class that extends the base Agent class.

A behaviour represents a task that an agent can carry out and is implemented as an object of a class that extends jade.core.behaviours.Behaviour. In order to make an agent execute the task implemented by a behaviour object it is sufficient to add the behaviour to the agent by means of the addBehaviour() method of the Agent class. Each class extending Behaviour must implement the action() method, that actually defines the operations to be performed when the behaviour is in execution and the done() method (returns a boolean value), that specifies whether or not a behaviour has completed and have to be removed from the pool of behaviours an agent is carrying out. Scheduling of behaviours in an agent is not pre-emptive (as for Java threads) but cooperative. This means that when a behaviour is scheduled for execution its action() method is called and runs until it returns. The termination value of a behaviour is returned by his onEnd() method [11].

Agent behaviours can be described as finite state machines, keeping their whole state in their instance variables. When dealing with complex agent behaviours, using explicit state variables can be cumbersome; so JADE also supports a compositional technique to build more complex behaviours out of simpler ones.

The FSMBehaviour is such a subclass that executes its children according to a Finite State Machine (FSM) defined by the user. More in details each child represents the activity to be performed within a state of the FSM and the user can define the transitions between the states of the FSM. When the child corresponding to state S_i completes, its termination value (as returned by the onEnd() method) is used to select the transition to fire and a new state S_j is reached. At next round the child corresponding to S_j will be executed. Some of the children of an FSMBehaviour can be registered as final states. The FSMBehaviour terminates after the completion of one of these children.

The following methods are needed in order to properly define a FSMBehaviour:

- public void **registerFirstState**(Behaviour state, java.lang.String name)
  Is used to register a single Behaviour *state* as the initial state of the FSM with the name *name*.
- public void **registerLastState**(Behaviour state, java.lang.String name)
  Is called to register one or more Behaviours as the final states of the FSM.
- public void **registerState**(Behaviour state, java.lang.String name)
  Register one or more Behaviours as the intermediate states of the FSM.
- public void **registerTransition**(java.lang.String s1, java.lang.String s2, int event)

For the state s1 of the FSM, register the transition to the state s2, fired by terminating event of the state s1 (the value of terminating event is returned by onEnd() method, called when leaving the state s1 - sub-behaviour s1 has completed).

- public void **registerDefaultTransition**(java.lang.String s1, java.lang.String s2)
  This method is useful in order to register a default transition from a state to another state independently on the termination event of the source state.

3.2 Using ATL for verification of a JADE agent

Since testing is based on observing only subset of all possible instances of system behaviour, it can never be complete.

Because testing and simulation can give us only confidence in the implementation of a software system, but cannot prove that all bugs have been found, we will use a formal method, the ATL model checking, for detecting and eliminating bugs in the design of a FSM - driven behaviour of a JADE agent.

Design validation using ATL involves checking whether a system design satisfies the system requirements expressed by ATL formulas.

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

1. In parallel with construction of the Finite State Machine of the FSMBehaviour, the corresponding ATL model is built.
2. The specification (ATL formula) representing a desired behavioural property is verified to hold for the model describing the FSMBehaviour.

By using our ATL Library [9] to perform ATL model checking, we can detect the states of the model where the ATL formula does not hold and then we can correct the given model or design by reviewing the java code for construction of the Finite State Machine.

4 Using ATL Library for verification of JADE agents

For verification of JADE agents, we need to overwrite the standard methods of JADE FSMBehaviour such that building of the ATL model is accomplished in parallel with the definition of the FSM.

```
ATLGraphModel model = null;
Dictionary dict = null;

public void registerState(Behaviour b, String name){
      if (dict.get(name) == null) {
         lastVertex++;
         model.addNode(lastVertex, name);
         dict.put(name, lastVertex);
      }
      super.registerState(b, name);
}
public void registerLastState(Behaviour b, String name){
      String newName = name + ",*FINAL*";
      lastVertex++;
      model.addNode(lastVertex, newName);
      dict.put(name, lastVertex);
      super.registerLastState(b, name);
 }

public void registerTransition(java.lang.String s1, java.lang.String s2, int event) {
```

```
public void registerTransition(java.lang.String s1, java.lang.String s2, int event) {
      lastEdge++;
      int v1 = Integer.parseInt(dict.get(s1).toString());
      int v2 = Integer.parseInt(dict.get(s2).toString());
      model.addEdge(lastEdge, v1, v2, "<" + event + ">");
      super.registerTransition(s1, s2, event) ;
}
```

Fig. 2 Overwriting the standard methods of the FSMBehaviour

Using the ***checkFSM()*** method, the new class ATL_FSMBehaviour extends the functionality of the standard JADE class FSMBehaviour by adding ATL model checking capability (the ATL formula to be verified is submitted as a parameter):

```
public boolean checkFSM(String ATLFormula){

  //model.setServer(model.SERVER_LOCALHOST);
  model.setServer(model.SERVER_USEIT);

  model.setDebug(false);
  model.setCompressed(true);

  try {
     String result = model.checkModel(xmlModel, ATLFormula, "0");
     System.out.println(result);

     int[] states = model.getStates();

     if (states == null) {
        System.out.println("There are no states in which given ATL formula is satisfied.");
        System.out.println("The FSM is not well defined!");
        return false;
     } else {
        System.out.println("The ATL formula is satisfied in the following states:");
        for (int i = 0; i < states.length; i++) {
           System.out.print(" " + Integer.toString(states[i]));
        }
        if (states.length != lastVertex + 1){
           System.out.println("\nThe FSM is not well defined!");
           return false;
        }
        else
        {
           System.out.println("\nThe FSM is well defined.");
           return true;
        }
     }
  } catch (ATLException ex) {
     System.out.println(ex.getError());
     return false;
  }
}
```

Fig. 3 A new method of FSMBehaviour for verification of an ATL formula

In the following example, the ATL formula checked is:

$$<< A >> @\, d \tag{2}$$

Thus, we verify that the state *d* is a reachable state.

In figure 4 is presented the underlying ATL model of the FSMBehaviour and loaded in ATL Designer:

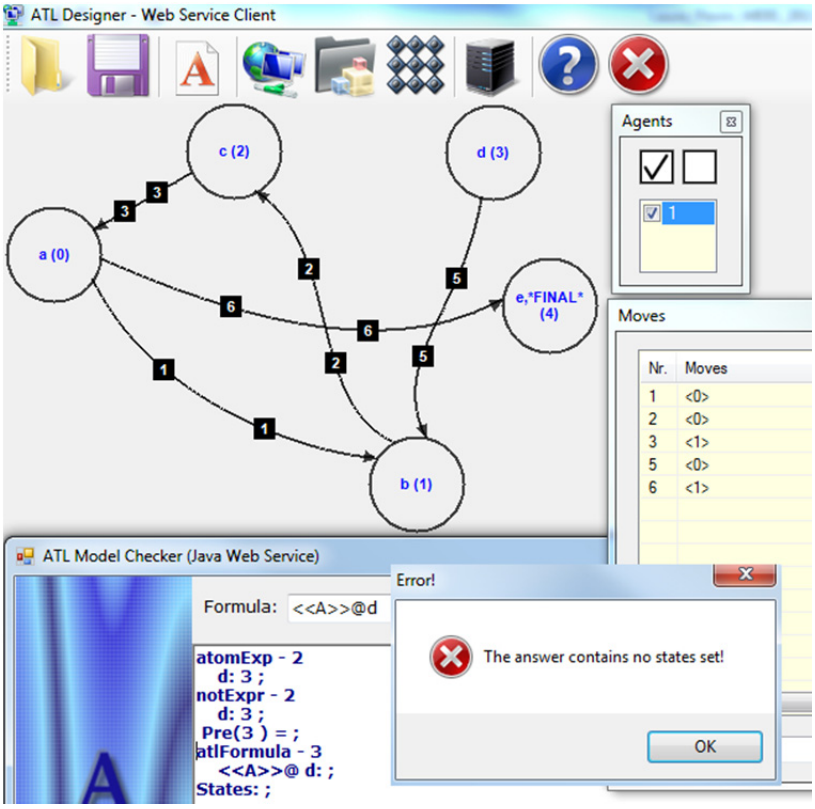

Fig. 4 Checking the ATL model in ATL Designer

As we can see from the figure 5, the desired behavioural property expressed by ATL formula does not hold and in conclusion the state *d* is not reachable.

The FSMBehaviour of the JADE agent must be clearly revised, so the new model to satisfy the specification expressed by ATL formula (2).

```
INFO: ---------------------------------------
Agent container Container-18@Server1 is ready.
---------------------------------------------
States: ;

There are no states in which given ATL formula is satisfied.
The FSM is not well defined!
```

Fig. 5 The ATL formula is not verified in the model

5 Conclusions

In this paper Alternating-time Temporal Logic was used for the automated verification of JADE agents. The proposed method can be applied to any software system (written in Java or C#) for which can be constructed an ATL model using methods of our ATL Library.

The software components (libraries, examples of code, web services, designers) of the ATL model checker tool used in this paper can be downloaded from *http://use-it.ro*.

References

[1] K.Y. Rozier, Survey: Linear Temporal Logic Symbolic Model Checking, *Computer Science Review*, Volume 5 Issue 2, 163-203, 2011
[2] J. Barnat, L. Brim, P. Ročkai, Scalable Shared Memory LTL Model Checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):139-153, 2010
[3] R. Alur, T. A. Henzinger, O. Kupferman, Alternating-time temporal logic, *Journal of the ACM*, 49(5), pp. 672–713, 2002.
[4] M. Kacprzak, W. Penczek, Fully symbolic Unbounded Model Checking for Alternating-time Temporal Logic, Journal Autonomous Agents and Multi-Agent System, Volume 11 Issue 1, pp. 69 – 89, 2005
[5] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, S. Tasiran, Mocha: modularity in model checking, in Proc. of CAV 98, volume 1427 of Lect. Notes in Comp. Sci., pp. 521-525. Springer-Verlag, 1998
[6] A. Lomuscio, F. Raimondi, Mcmas: A model checker for multi-agent systems, in Proc. of TACAS 06, volume 3920 of Lect. Notes in Comp. Sci., pp. 450-454, Springer-Verlag, 2006.
[7] L. F. Cacovean, F. Stoica, D. Simian, A New Model Checking Tool, *Proceedings of the 5th European Computing Conference (ECC '11)*, Paris, France, April 28-30, 2011
[8] W. van der Hoek, M. Wooldridge, Tractable multiagent planning for epistemic goals, in *Proceedings of AAMAS-02*, pp. 1167-1174, ACM Press, 2002.
[9] L. F. Cacovean, F. Stoica, WebCheck – ATL/CTL model checker tool, http://use-it.ro
[10] Java Agent Development Framework (JADE), http://jade.tilab.com/
[11] F. Bellifemine, G. Caire, T. Trucco, G. Rimassa, JADE programmer's guide, http://jade.tilab.com, 2013

Laura Florentina STOICA
Faculty of Science,
"Lucian Blaga" University
Department of Mathematics and Informatics
5-7 Dr. Ratiu Street, Sibiu
ROMANIA
E-mail: laura.cacovean@ulbsibiu.ro

Florin STOICA
Faculty of Science,
"Lucian Blaga" University
Department of Mathematics and Informatics
5-7 Dr. Ratiu Street, Sibiu
ROMANIA
E-mail: florin.stoica@ ulbsibiu.ro

Mircea Florian BOIAN
Faculty of Mathematics and Computer Science, "Babes Bolyai" University
Department of Computer Science
1 M. Kogalniceanu Street, Cluj Napoca
ROMANIA
E-mail: florin@cs.ubbcluj.ro