INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
ISSN 1841-9836, 10(5):718-731, October, 2015.

# Verification of JADE Agents Using ATL Model Checking

L.F. Stoica, F. Stoica, F.M. Boian

**Laura Florentina Stoica*, Florin Stoica**
1. Department of Mathematics and Informatics,
Faculty of Science, "Lucian Blaga" University,
5-7 Dr. Ratiu Street, Sibiu, Romania
2. Center of Scientific Research in Informatics and Information Technology
"Lucian Blaga" University
5-7 Dr. Ratiu Street, Sibiu, Romania
laura.cacovean@ulbsibiu.ro, florin.stoica@ulbsibiu.ro
* Corresponding author: laura.cacovean@ulbsibiu.ro

**Florian Mircea Boian**
Department of Computer Science,
Faculty of Mathematics and Computer Science, "Babes-Bolyai" University,
1 M. Kogalniceanu Street, Cluj Napoca, Romania,
florin@cs.ubbcluj.ro

**Abstract:**
It is widely accepted that the key to successfully developing a system is to produce a thorough system specification and design. This task requires an appropriate formal method and a suitable tool to determine whether or not an implementation conforms to the specifications. In this paper we present an advanced technique to analyse, design and debug JADE software agents, using Alternating-time Temporal Logic (ATL) which is interpreted over concurrent game structures, considered as natural models for compositions of open systems. In development of the proposed solution, we will use our original ATL model checker. In contrast to previous approaches, our tool permits an interactive or programmatic design of the ATL models as state-transition graphs, and is based on client/server architecture: ATL Designer, the client tool, allows an interactive construction of the concurrent game structures as a directed multi-graphs and the ATL Checker, the core of our tool, represents the server part and is published as Web service.
**Keywords:** model checking, ATL, agents, JADE, FSM

## 1 Introduction

Because of competition on the information technology market, the development of the information systems must be achieved with maximum productivity. In many cases, the price paid is the diminution of quality of software products.

The aim of the methods of software engineering is to increase the quality and reliability of software. In particular, the formal methods are focused on reliability and correctness, using a mathematical support.

The reliability of a system can be increased through modelling of the system before the implementation of code, followed by the validation and verification of its correctness.

Validation is accomplished with respect to informal requirements of the system. The designer checks if the model reflects the expected behaviour of the system.

Verification is accomplished with respect to system specifications expressed in a formal manner. A formal specification is an abstract model of the system, expressed in a formal notation.

Verification of a software system involves checking whether the system in question behaves as it was designed to behave. Design validation involves checking whether a system design

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

Model checking is particularly well-suited for the automated verification of finite-state systems, both for software and for hardware. Main concern of formal methods in general, and model checking in particular, is helping to design correct systems [2]. Detecting and eliminating bugs as early in the design cycle as possible is clearly an economic imperative. For example, the Pentium FDIV bug (a bug in the Intel P5 Pentium floating point unit discovered in 1994) cost Intel Corporation a half billion dollars.

Model checking is 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]. ATL is also widely used to reason about strategies in multiplayer games. The semantics of ATL is formalized by defining games such that the satisfaction of an ATL formula corresponds to the existence of a winning strategy.

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

ATL defines "cooperation modalities", of the form $\langle\langle \mathcal{A} \rangle\rangle \varphi$, where $\mathcal{A}$ is a group of agents. The intended interpretation of the ATL formula $\langle\langle \mathcal{A} \rangle\rangle \varphi$ is that the agents $\mathcal{A}$ can cooperate to ensure that $\varphi$ holds (equivalently, that $\mathcal{A}$ have a winning strategy for $\varphi$) [4].

ATL has been implemented in several symbolic 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. An ISPL file fully describes a multi-agent system (both the agents and the environment) [6].

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

Symbolic model checkers analyse the state space symbolically using Ordered Binary Decision Diagrams (OBDDs), which are data structures for representing Boolean functions and were introduced in [12].

In [13], [14], [15], [16] are presented comparisons between symbolic and explicit model checking of software or hardware systems. For most hardware designs which are based on a clocked-approach and thus are synchronous, the symbolic model checking approach is more appropriate [14]. The explicit-state model-checkers performs better in case of using large states needed to include some information [13], also for nondeterministic, high-level models of hardware protocols and for model checking of concurrent asynchronous software systems [15], [16].

In [7] is presented our tool, a new interactive ATL model checker environment based on algebraic approach. The original implementation of the model checking algorithm is based on Relational Algebra expressions translated into SQL queries. 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.

Taking into account the above considerations, in our tool we are using an explicit-state model technique. Thus, in contrast to previous approaches, our 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 $\varphi$ of a given ATL model to set of nodes over which formula $\varphi$ is satisfied. The implementation of the model checking algorithm is based on Java code generated by ANTLR (Another Tool for Language Recognition) using an original ATL grammar and provides error-handling for eventual lexical/syntax errors in formula to be analysed. 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, using components of our tool, we will show how ATL model checking technology can be used for automated verification of multi-agent systems, developed with JADE.

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

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

Using Java version of the ATL Library - a component of our ATL model checker, used for development of custom applications with large ATL models - into JADE agents is inserted the code necessary to build in a transparent manner the ATL model which will be verified at runtime.

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 outlined the architecture of our ATL model checker and is made a brief analysis of its performance. In section 4 is presented the JADE FSMBehaviour and formal models used to build an equivalent concurrent game structure. These concepts are applied in section 5 where ATL Library is used to verify the design of the JADE agents having FSM - driven behaviours. Conclusions are presented in section 6.

## 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 $\Lambda = \{1, ..., k\}$;
- $Q$ denotes the finite *set of states* ;
- $\Gamma$ denotes the finite *set of propositions* (or *observables*);
- $\gamma : Q \to 2^{\Gamma}$ 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* $d : \Lambda \times Q \to 2^M$ associates for each player $a \in \{1, ..., k\}$ 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 $\langle j_1, ..., j_k \rangle$ such that $j_a \in d_a(q)$ for each player $a \in \Lambda$, represents a *move vector* at $q$.

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

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

## 2.2 Syntax of ATL

The ATL operator $\langle\langle\rangle\rangle$ is *a path quantifier*, parameterized by sets of agents from $\Lambda$. The operators $\bigcirc$ ('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 $\varphi$.

The temporal logic ATL is defined with respect to a finite set of agents $\Lambda$ and a finite set $\Gamma$ 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 $\varphi$, $\varphi_1$ and $\varphi_2$ are ATL formulas;
3. $\langle\langle\mathcal{A}\rangle\rangle \bigcirc \varphi$, $\langle\langle\mathcal{A}\rangle\rangle\Box\varphi$, $\langle\langle\mathcal{A}\rangle\rangle\Diamond\varphi$ or $\langle\langle\mathcal{A}\rangle\rangle\varphi_1 U \varphi_2$, where $\mathcal{A} \subseteq \Lambda$ is a set of players, and $\varphi$, $\varphi_1$ and $\varphi_2$ 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 $\Lambda = \{1, ..., k\}$ 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 $f_a : Q^+ \to D_a$ that maps every nonempty finite state sequence $\lambda = q_0 q_1 ... q_n$, $n \geq 0$, 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 $\lambda$ of a computation a move $f_a(\lambda)$ for player $a$ in the last state of $\lambda$.

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

- $j_a = f_a(\lambda[0, i])$ for all players $a \in \mathcal{A}$, and

- $\delta(q_i, j_1, ..., j_k) = q_{i+1}$.

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

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

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

## 3 Architecture, scalability and performance of the ATL model checker

Our ATL model checker tool contains the following packages:

- *ATL Compiler* - the core of our tool, embedded into a Web Service (ATL Checker);
- *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).

ATL Designer [9] implements the Tic-Tac-Toe (TTT) game, using an algorithm which looks for infallible conditional plans to achieve a winning strategy that can be defined via ATL formulae. The game is played by two opponents with a turn-based modality on a $3 \times 3$ board. The two players take turns to put pieces on the board. A single piece is put for each turn and a piece once put does not move. A player wins the game by first lining three of his or her pieces in a straight line, no matter horizontal, vertical or diagonal. A user can play TTT game against the computer and the ATL model checking algorithm is used to achieve a winning strategy for the computer.

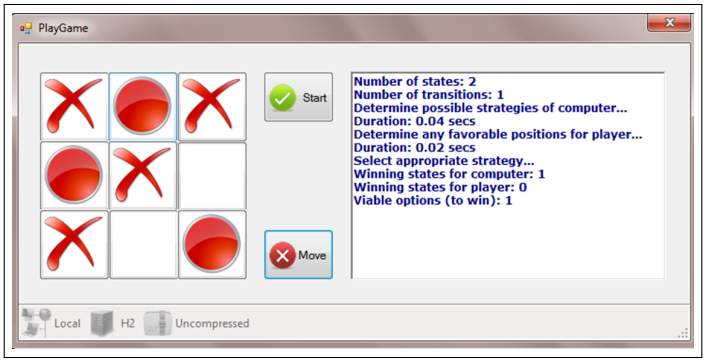


Figure 1: The computer has found a winning strategy and won a game against the human user

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

For a set $\mathcal{A}$ of agents, the implementation of most ATL operators in the model checking algorithm [3] implies the computation of function $Pre(\mathcal{A}, \Theta)$, where $\Theta \subseteq Q$. The value returned by $Pre(\mathcal{A}, \Theta)$ represents the set of states from which agents $\mathcal{A}$ can enforce the system into some state in $\Theta$ in one move.

In [7] we made a implementation of the function $Pre()$ using $SQL$ statements, ready to be executed on a high-speed database server. Our approach is to use $SQL$ and its massive scalability features in verification of large real-world systems.

In order to analyze their impact in the performance of the ATL model checker, were used three different database servers necessary to determine the winning strategy in the Tic-Tac-Toe game, namely *MySql 5.5*, *H2 1.3* and respectively *Microsoft SQL Server 2008* using an *Intel Core I5*, *2.5 GHz*, *4Gb RAM*.

Benchmark results are presented in figure 2. Results from [18] showed that both $SMV$ (a symbolic CTL model checker) and $SPIN$ (a well-known explicit-state LTL model checker tool) were able to find an optimal strategy for a player in less than one second, on a $3 \times 3$ board. As we can see from figure 2, our ATL model checker tool is not as fast as the CTL/LTL tools, but we must take into consideration that an ATL model is more expressive (with ATL we can quantify over the individual powers of one player or a cooperating team of players, ATL models capture various notions of synchronous and asynchronous interaction between open systems, etc.). But our tool achieves substantial speedup over an implementation of the Tic-Tac-Toe in the Reactive Modules Language ($RML$) of the ATL model checker MOCHA [17], [7].

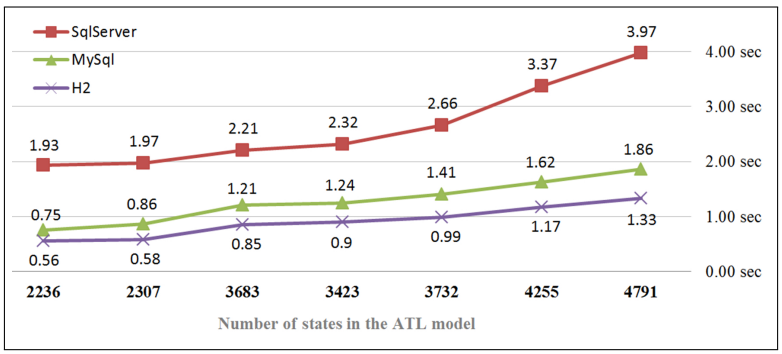


Figure 2: The performance of ATL model checker related to database server used

# 4 Using ATL model checking in agent-based systems

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

The domain of software agents being relatively recent and in a continuous development, there is no a standard definition, unanimous accepted definition for an agent.

However, the autonomy is the central property of agent concept, others properties having a different importance, regarding of concrete considered applications. We will consider that an agent is a computational system situated in a runtime environment and capable of autonomous actions in that environment, in order to accomplish his planned objectives.

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

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.

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

## 4.1 JADE agents with FSM behaviours

JADE 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.

The computational model of an agent is multitask, where tasks (or behaviours) are executed concurrently. A scheduler, internal to the base Agent class and hidden to the programmer, automatically manages the scheduling of behaviours.

A behaviour represents a task that an agent can carry out and is implemented as an object of a class that extends the standard JADE class 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 which 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]. The path of execution of the agent thread is showed in the following pseudocode:

```
void AgentLifeCycle() {
  setup();
  while (true) {
    if (was called doDelete()) {
      takeDown();
      return;
    }
    Behaviour b =
    getNextActiveBehaviourFromSchedulingQueue();
    b. action();
    if (b.done() returns true) {
      removeBehaviourFromTheSchedulingQueue (b);
      int terminationValueOfTheBehaviour =  b.onEnd();
    }
  }
}
```

Behaviours work just like co-operative threads, but there is no stack to be saved. Therefore, the whole computation state must be maintained in instance variables of the Behaviour and its associated Agent.

Following this idiom, 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 JADE abstract class CompositeBehaviour provides the possibility of combining simple behaviours together (children) to create complex behaviours. The actual operations performed by executing this behaviour are defined inside its children while the composite behaviour deals with execution planning. The scheduling policy must be defined by subclasses of CompositeBehaviour.

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.

### 4.2 A formal model of the FSMBehaviour

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

A JADE finite state machine is a tuple $FSM = (Q_{FSM}, \Pi, \pi, q_0, F, t, \delta_{FSM})$ where:

- $Q_{FSM}$ is a finite, non-empty *set of states*;
- $\Pi$ denotes the finite *set of state names*;
- $\pi : Q_{FSM} \to \Pi$ is called the *labelling function*, defined as follows: for each state $q \in Q_{FSM}$, $\pi(q) \in \Pi$ is the *name of state* $q$;
- $q_0$ is an element of $Q_{FSM}$, the *initial state*;
- $F \subseteq Q_{FSM}$ is the *set of final states*;
- $t : Q_{FSM} \to 2^{\mathbb{Z} \cup \{default\}}$ is called the *terminating function*, where for each state $q \in Q_{FSM}$, $t(q) \subseteq \mathbb{Z} \cup \{default\}$ represents the set of admissible termination codes of the state $q$;
- The transition function $\delta_{FSM}(q, j)$, associates to each state $q \in Q_{FSM}$ and each termination code $j$ of $q$ the state that results from state $q$ if the child behaviour associated with the state $q$ returns at finish the value $j$.

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

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

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

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

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

### 4.3 ATL model of the FSMBehaviour

For a JADE finite state machine defined in section 4.2, the equivalent concurrent game structure $S = \langle \Lambda, Q, \Gamma, \gamma, M, d, \delta \rangle$ is defined as follows:

- There is only one agent, i.e. $\Lambda = \{1\}$;

- The set of states is $Q = Q_{FSM}$;
- The finite *set of propositions* is defined by $\Gamma = \Pi \cup \{*FINAL*\}$;
- The *labelling function* $\gamma : Q \to 2^{\Gamma}$ is defined as follows:

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

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

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

- The *alternative moves function* $d : \Lambda \times Q \to 2^M$ is defined by $d(1, q) = t(q)\ \forall q \in Q$
- The *transition function* $\delta$ is defines as follows: $\delta(q, \langle j \rangle) = \delta_{FSM}(q, j)\ \forall q \in Q\ and\ \forall j \in t(q)$.

### 4.4 Using ATL for verification of the FSM - driven behaviour of a JADE agent

Discrepancies between actual and expected results are called *conformance failures* and may indicate any of the following: implementation bug, modelling error, specification error or design error.

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. For the beginning, the corresponding ATL is constructed following rules described in section 4.3
2. Then, a given specification (ATL formula) representing a desired behavioural property is verified to hold for the model obtained at step 1.

Using ATL Library [9] to perform ATL model checking, we can detect error states (the states of the model where the ATL formula does not hold) and then we can correct the given model or design.

## 5 Using ATL Library to verify the design of JADE agents

Using the Java version of our ATL Library, the standard methods of JADE FSMBehaviour have been overwritten such that building of the ATL model to be done in parallel with the definition of the FSM.

The new class $ATL\_FSMBehaviour$ extends the functionality of the standard JADE class FSMBehaviour by adding ATL model checking capability:

```
public class MyAgent extends Agent {
  // State names
  private static final String STATE_A = "a";
  private static final String STATE_B = "b";
  private static final String STATE_C = "c";
  private static final String STATE_D = "d";
  private static final String STATE_E = "e";
  private boolean wellDefined = false;

  protected void setup() {
    ATL_FSMBehaviour fsm = new  ATL_FSMBehaviour(this);
    fsm.registerFirstState(new RandomGenerator(2), STATE_A);
    fsm.registerState(new NamePrinter(), STATE_B);
    fsm.registerState(new NamePrinter(), STATE_C);
    fsm.registerState(new NamePrinter(), STATE_D);
    fsm.registerLastState(new NamePrinter(), STATE_E);
    fsm.registerTransition(STATE_A, STATE_B, 0);
    fsm.registerTransition(STATE_A, STATE_E, 1);
    fsm.registerTransition(STATE_B, STATE_C, 0);
    fsm.registerTransition(STATE_C, STATE_D, 0);
    fsm.registerTransition(STATE_D, STATE_B, 0);
    wellDefined = fsm.checkFSM();
    System.out.print("Execution: ");
    addBehaviour(fsm);
    }
  }
```

In the above example, the child behaviour *NamePrinter* displays only the name of the parent state. The *RandomGenerator* behaviour allows FSM to randomly select the transition to fire from the parent state (the terminating event of the parent state is chosen randomly from the admissible values). The *checkFSM()* make calls in ATL Library to perform verification of the defined FSM.

In our example, the ATL formula checked is:

$$\langle\langle \mathcal{A} \rangle\rangle \Diamond \ (*FINAL*) \tag{2}$$

Thus, we verify that in every computation the agent will reach a final state.

In figure 3 is presented the underlying ATL model of the FSMBehaviour described in **MyAgent class** and loaded in ATL Designer:

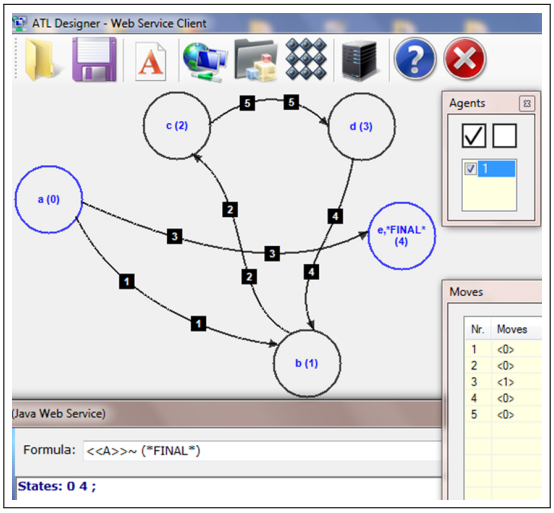


Figure 3: Checking the ATL model in ATL Designer

As we can see from the figure 3, the desired behavioural property expressed by ATL formula (2) does not hold for the entire model. The only states in which the formula holds are STATE_A (0) and STATE_E (4). This can lead to unexpected results at runtime, as we can see from the figure 4:

INFO:
Agent container Container-4@Server1
is ready.

States: 0 4 ;
The ATL formula is satisfied in the following states: 0 4
**The FSM is not well defined!**
Execution: ae
The end

Figure 4a: The final state is reached

INFO:
Agent container Container-1@Server1
is ready.

States: 0 4 ;
The ATL formula is satisfied in the following states: 0 4
**The FSM is not well defined!**
Execution:abcdbcdbcdbcdbcdbcdbcdbcdb
cdbcdbcdbcdbcdbcdbcdbcdbcdbcdbcdbcd
bcdbcdbcdbcdbcdbcdbcdbcd....

Figure 4b: The final state is not reached, there is an infinite loop!

Figure 4: Unexpected results at runtime

In the figure 4 is presented a FSM with a correct design, satisfying the specification (2):

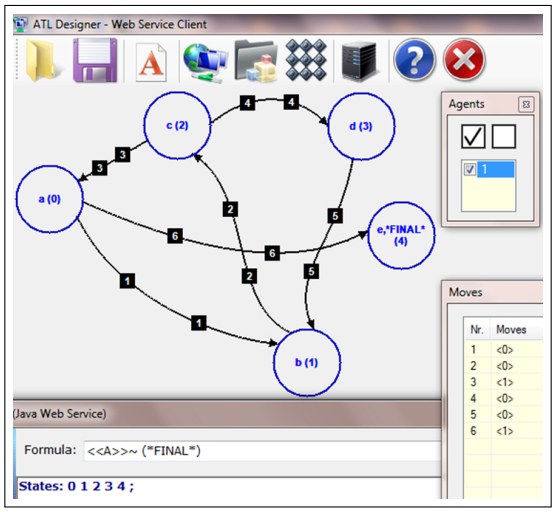


Figure 5: An revised ATL model in which the ATL formula (2) holds

In figure 6a. is presented the revised version of the JADE FSMBehaviour, so that the corresponding ATL model which was built automatically satisfies the specification (2). Figure 6b. shows some execution scenarios of the new JADE agent. Obviously, in all cases the final state will be reached and agent will terminate properly.

```
fsm.registerFirstState(
      new RandomGenerator(2), STATE_A);
fsm.registerState(new NamePrinter(), STATE_B);
fsm.registerState(
      new RandomGenerator(2), STATE_C);
fsm.registerState(new NamePrinter(), STATE_D);
fsm.registerLastState(
      new NamePrinter(), STATE_E);
fsm.registerTransition(STATE_A, STATE_B, 0);
fsm.registerTransition(STATE_A, STATE_E, 1);
fsm.registerTransition(STATE_B, STATE_C, 0);
fsm.registerTransition(STATE_C, STATE_D, 0);
fsm.registerTransition(
      STATE_C, STATE_A, 1);
fsm.registerTransition(STATE_D, STATE_B, 0);
```

Figure 6a: The revised code of the JADE agent

```
States: 0 1 2 3 4 ;
The ATL formula is satisfied in
the following states: 0 1 2 3 4
The FSM is well defined.
Execution: abcdbcae
The end
Execution: abcdbcabcdbcdbcae
The end
Execution: ae
The end
Execution: abcabcdbcdbcdbc
dbcabcdbcabcae
The end
```

Figure 6b: The verification of the ATL model ensure that in all scenarios the agent will behave correctly

## 6 Conclusions

In this paper Alternating-time Temporal Logic was used for the automated verification of software systems based on JADE agents with FSM - driven behaviours. Our solution is based on Java version of ATL Library, a component of our ATL model checker tool.

The proposed method can be easily adapted to any software system (written in Java or *C#*) capable to exhibit an ATL model (in the form of a concurrent game structure). The original classes (agents) can be extended with specific functionality in order to create an ATL model on-the-fly, while executing the regular code of the application. The ATL model thus constructed is verified against specifications expressed by ATL formulas, at debugging time. After the validation phase, the model checking functionality will be disabled in the release build of the software system.

Our ATL model checker tool is based on client/server architecture to address the time constraints in verification of large models. By using a database-based technology in the core of the ATL model checker, our tool provides a good foundation for further improvement of its performance and scalability.

On the client side, ATL Designer allows an interactive graphical specification of an ATL model as a directed multi-graph. Programmatic construction of a large ATL model is made using ATL Library, available for *C#* and Java languages by now, but we intend to provide interfaces for other popular programming languages.

The server component of our tool (ATL Checker) was published as a Web service, exposing its functionality through standard interfaces. By using Web Services a client can perform model checking without the need of configuring servers.

All components of the new ATL model checker tool can be downloaded from `http://use-it.ro`.

Further investigation on improving performance in verification of ATL formulas will be done through parallelized SQL queries using the horizontal scalability features such as Parallel Pipelined Table Functions (PTF) provided by Oracle databases.

## Bibliography

[1] K.Y. Rozier (2011); Survey: Linear Temporal Logic Symbolic Model Checking, *Computer Science Review*, vol. 5, Issue 2: 163-203.

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

[3] R. Alur, T.A. Henzinger, O. Kupferman (2002); *Alternating-time temporal logic, Journal of the ACM*, 49(5): 672-713.

[4] M. Kacprzak, W.Penczek (2005); Fully symbolic Unbounded Model Checking for Alternating-time Temporal Logic, *Journal Autonomous Agents and Multi-Agent System*, 11(1): 69-89.

[5] R. Alur et al (1998); Mocha: modularity in model checking, in Proc. Of CAV 98, vol. 1427 of *Lect. Notes in Comp. Sci., Springer-Verlag*: 521-525.

[6] A. Lomuscio, F.Raimondi (2006); Mcmas: A model checker for multi-agent systems, in Proc. of TACAS 06, vol. 3920 of *Lect. Notes in Comp. Sci., Springer-Verlag*: 450-454.

[7] F. Stoica, L.F. Cacovean (2014); Implementing an ATL Model Checker tool using Relational Algebra concepts, in *Proceeding The 22th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*: 361-366.

[8] W. Van der Hoek, M. Wooldridge (2002); Tractable multiagent planning for epistemic goals, in *Proceedings of AAMAS-02, ACM Press*: 1167-1174.

[9] L.F. Stoica, 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 (2013); JADE programmer's guide, http://jade.tilab.com

[12] R. E. Bryant (1986); Graph-based algorithms for boolean function manipulation, *IEEE Transactions on Computers*, C-35(8): 677-691.

[13] C. Eisner, D. Peled (2002); Comparing Symbolic and Explicit Model Checking of a Software System, In *Proc. SPIN Workshop on Model Checking of Software*, volume 2318 of LNCS, Volume 55: 230-239.

[14] F. Lerda, N. Sinha, M. Theobald (2003); Symbolic Model Checking of Software, *Electronic Notes in Theoretical Computer Science*, 89(3): 480-498.

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

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

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

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