Modeling and Verifying Agent-based Communities of Web Services

Wei Wan[1], Jamal Bentahar[2], and Abdessamad Ben Hamza[2]

[1]Department of Electrical and Computer Engineering, Concordia University
[2]Concordia Institute for Information Systems Engineering, Concordia University
w_wan@encs.concordia.ca,{bentahar,hamza}@ciise.concordia.ca,

Abstract. Communities of web services are virtual spaces that can dynamically gather different web services having complementary functionalities in order to provide composite services with high quality. In the last two years, some approaches have been proposed using multi-agent systems to organize communities of web services. This trend has increased the flexibility but also the system complexity. These systems are hard to check by simply inspecting their models. Therefore, model checking, which is a well-established formal technique for verifying communication and cooperation in multi-agent systems, is used in this paper to verify the system correctness in terms of satisfying desirable properties. The approach presented in the paper is used to verify these communities modeled in UML activity diagram. We first translate the activity diagram into an interpreted system model using predefined transformation rules. Specifications are expressed as formulae in a logic extending the Computation Tree Logic CTL^* with agent commitments needed for their communication. Then, both the model and formulae are used as inputs for the multi-agent symbolic model checker MCMAS. We illustrate our approach with a short case study, in which we show how communication properties of simulated communities are verified.

Key words: Multi-Agent Systems, Communities of Web Services, UML, Model Checking, MCMAS

1 Introduction

As the internet becomes more and more prevalent, communities of web services, which are larger scale virtual network societies integrated web services, attract more and more attention [1, 2]. In the last two years, these communities started to move toward "agent-like" models that include largely independent agent-based web services. However, this merging results in more complex systems where functional and non-functional properties cannot be easily checked by simply inspecting the system model. Since it is very expensive to modify communities of web services that have been deployed, it is desirable to have methods available for the verification of communities' properties earlier in the design phases. Model checking [5] is a suitable solution in this case because it

is a formal technique allowing the automatic verification of the systems design against specific properties that capture the requirements.

In nineties, Researchers have put forward in [7] an application of model checking within the context of the logic of knowledge. After that, several approaches have been proposed for model checking multi-agent systems. In [16], Wooldridge et al. have proposed an imperative programming language, MABLE to specify multi-agent systems along with a Belief-Desire-Intention (BDI) logic to express the properties. SPIN, an automata-based model checker has been used to verify if the specified MABLE model satisfies the expressed properties. Another method based on the SPIN model checker has been developed in [3] using AgentSpeak(F) Language, a BDI logic-based programming language [12]. As a model grows, automata-based model checking can face a serious state explosion problem. One technique to avoid this problem is symbolic model checking based on Ordered Binary Decision Diagrams (OBDDs). NuSMV [4], MCK [14] and MCMAS [9] are examples of model checkers using this approach. NuSMV supports both Linear Temporal Logic (LTL) and branching time logic (CTL). MCK works on a particular input model of synchronous interpreted systems of knowledge. The specification formulae in MCK can be either LTL or CTL augmented with knowledge. Similar to MCK, in MCMAS, models are described into a modular language called Interpreted Systems Programming Language (ISPL). Although the framework of interpreted systems is powerful and popular in multi-agent systems, it cannot be directly used by designers to describe business and industrial systems. For this type of applications, it deems appropriate to use suitable modeling languages such as UML (Unified Modeling Language).

The motivation of this paper is to build the connection among UML, agent-based communities properties, and symbolic model checking so that we can use exist model checker, like MCMAS, to check these communities' models directly. We propose an approach based upon symbolic model checking to verify communities presented by UML activity diagram, which shows the activities and flow of control for the model [15]. We formalize agent-based communities of web services with the execution semantics of UML activity diagram. We adopt CTL^{*CA} proposed in [2] for communicating agents as the logic for specifying the properties to be checked. We use the MCMAS model checker in our symbolic approach for the verification of communities of web services. There are two reasons behind choosing MCMAS: 1) unlike NuSMV and MCK, MCMAS supports directly agent specifications we need for agent-based communities of web services; and 2) in terms of the adopted specification language, MCMAS is the closest to CTL^{*CA}. We experiment this approach with an implementation to verify the $PNAWS$ protocol (Persuasion/Negotiation protocol for Agent-based Web Services) [2]. $PNAWS$ is a communication protocol used by agent-based web services to negotiate joining a given community.

The structure of this paper is as follows: In Section 2, we present an overview of our model checking approach and explain how we formalize the activity diagram to represent communicating agent-based web services. The specification language CTL^{*CA} logic for communicating agents will be also discussed. In Sec-

tion 3, we define the rules for mapping and transforming formalized activity diagram model and properties specification into the Interpreted Systems Programming Language (ISPL), which is used as the input language of the MCMAS model checker. Section 4 presents the experimental results of verifying the $PNAWS$ protocol with MCMAS. Finally, we summarize our work and discuss future works in Section 5.

2 Modeling / Specifying Communities of Web Services

2.1 Approach Overview

Model checking is a three-step process [5]: modeling, specification, and verification. We use UML activity diagrams to model the system, CTL^{*CA} as specification language to state the properties that the design must satisfy, and symbolic model checking with OBDDs for verification.

Fig.1 illustrates our general approach. It starts with modeling communities of web services as activity diagrams and specifying the properties as formal requirements. The modeled system and formalized specifications are read as inputs by our automatic transformation engine, which uses transformation definitions (rules) to map the input model and specifications (properties) into the ISPL model and formulae. Finally, the MCMAS model checker verifies the ISPL model against the formulae. Witnesses are generated if the formulae are true (i.e. the properties are satisfied); otherwise, counterexamples are generated.

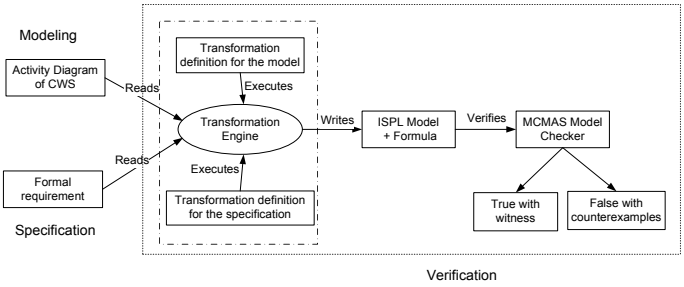

Fig. 1. Structure of model checking communities of web services (CWSs)

Before using this approach, an issue should be resolved: the activity diagram modeling the community of web services can have an infinite state space, while symbolic model checking requires the state space to be finite. To convert activity diagram from infinite to finite state space, Eshuis and Wieringa have proposed in [6] some techniques to remove unbounded states, which do not have a maximum number of their *active instances*. We adopt this method in our approach.

2.2 UML Activity Diagrams

A UML activity diagram shows the activities and flow of control for the model [15]. Activity states are represented with rounded rectangles, a black solid circle

stands for an initial node and a black in-out circle is a final node. A diamond represents a decision or merging state. A bar shows an activity that splits a flow into several concurrent flows or an activity that synchronizes several concurrent flows and joins them into one single flow.

Fig.2 shows an activity diagram for a concrete example of $PNAWS$ protocol model [2]. According to this protocol, agent-based web services interact with each other in a negotiation setting. Our example diagram presents the Master web service (MWS) agent's behavior of inviting a Slave web service to join its community and the Slave web service (SWS) agent's behavior of negotiating the joining contract. The MWS, which represents the community, starts the session by sending the invitation. The SWS can either accept or refuse. If the SWS accepts, the session will end with successful invitation. Otherwise, the MWS will defend the invitation proposal with negotiation arguments. Then, if the defence is not definitely accepted or refused, the MWS and SWS will enter a negotiation process consisting of a sequence of challenge/justification and attack until they achieve an agreement, refusal or timeout.

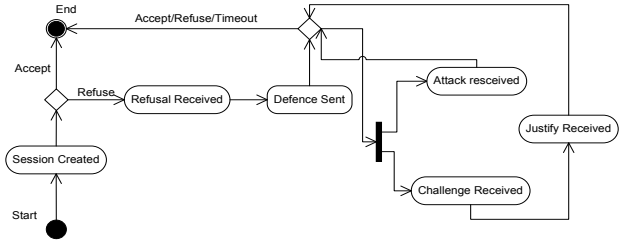

Fig. 2. Activity diagram of $PNAWS$ protocol

The activity diagram used in this paper describes the behavior of agents that interact with each other. These agents perform certain actions according to the protocol they use, which is a set of rules describing the allowed communicative acts in different situations. An agent has beliefs, goals, and intentions that are stored in a database accessible to the agent but external to the activity diagram. Activity states represent activities preformed by certain agents, such as accepting or refusing a proposal. A transition from one state to another is trigged by a set of internal (by agent's own actions) or external (by other agents' actions) activities. We use $\xrightarrow{Act}$ to represent the transition relation.

In order to use activity diagrams in our symbolic model checking approach, we need to define their formal semantics. Because they are associated to agent communication protocols in our approach, a suitable solution would be to define a formal *agent hypergraph* from the notion of *activity hypergraph* used to model check activity diagrams [6]. The idea behind an *agent hypergraph* is to capture the execution structure of the communication protocol among agent-based web services. We use CTL^{*CA} model to represent the communicative acts agents use when communicating. These communicative acts are defined as action performed on public commitments the agents make. For example, by inviting a Slave web service to join a community, the Master *creates* a new public commitment and

accepting this invitation means *accepting* the content of this public commitment. An *agent hypergraph* is defined as a tuple: $< S, s_0, Ag, Act, \xrightarrow{Act}, V_{PC} >$, where:

- S is a set of all possible states in the system. There are three kinds of states in this set: one initial state, at least one final state, and none or several activity states which are not initial state or final states.
- s_0 is the initial state.
- Ag is a non-empty set of agents.
- Act is a set of allowed actions agents can perform.
- $\xrightarrow{Act} \subseteq S \times Ag \times S$ is the transition relation. We write $s_i, Ag_n \xrightarrow{Act_k} s_j$ to express how the agent Ag_n evolves from one state s_i to another state s_j by performing the action Act_k.
- $V_{PC} : S \rightarrow 2^C$ is a function associating to each state the set of public commitments made in this state, where C is the set of all public commitments.

2.3 Logic for Specification - CTL^{*CA}

Syntax We use CTL^{*CA} [2] to specify the properties our agent-based communities of web services should satisfy. CTL^{*CA} extends CTL^* [5] by adding public commitments and action formulae. This logic supports two kinds of formulae: state formulae $\mathcal{S}$ evaluated over states and path formulae $\mathcal{P}$ evaluated over paths that are infinite sequences of states. We use $p, p_1, p_2, \ldots$ to range over the set of atomic propositions Φ_p and $\phi_1, \phi_2, \ldots$ to range over path formulae. The syntax of this logic is given in Table 1.

Table 1. The Syntax of CTL^{*CA} Logic

$\mathcal{S} ::= p \mid \neg\mathcal{S} \mid S \vee \mathcal{S} \mid A\mathcal{P} \mid E\mathcal{P} \mid PC(Ag_1, Ag_2, t, \mathcal{P})$
$\mathcal{P} ::= \mathcal{S} \mid \mathcal{P} \vee \mathcal{P} \mid X^+\mathcal{P} \mid X^-\mathcal{P} \mid \mathcal{P}U^+\mathcal{P} \mid \mathcal{P}U^-\mathcal{P} \mid \mathcal{P} \therefore \mathcal{P}$
$\mid Act_k(Ag_n, PC(Ag_1, Ag_2, t, \mathcal{P}))$

The temporal operator $X^+\phi_1$ means in the next state ϕ_1 is true, $X^-\phi_1$ means in the previous state ϕ_1 is true, $\phi_1 U^+ \phi_2$ means ϕ_1 is true utile ϕ_2 becomes true and $\phi_1 U^- \phi_2$ means ϕ_1 is rue since ϕ_2 was true. A stands for the universal path quantifier and E stands for the existential path quantifier. The formula $\phi_1 \therefore \phi_2$ means that ϕ_1 is an argument for ϕ_2 and is read as: ϕ_1 so ϕ_2. This operator introduces argumentation as a logical relation between path formulae.

The formula $PC(Ag_1, Ag_2, t, \phi_1)$ is the public commitment made by agent Ag_1 at the moment t towards agent Ag_2 saying that the path formula ϕ_1 is true. $Act_k(Ag_n, PC(Ag_1, Ag_2, t, \phi_1))$ means that agent Ag_n ($n \in \{1, 2\}$) performs an action Act_k on the commitment made by Ag_1 towards Ag_2. The set of actions performed on commitments may change to suit different systems. For communities of web services, we use $Create$, $Accept$, $Refuse$, $Defend$, $Challenge$, $Justify$, and $Attack$.

Formal Semantics The formal model M associated to this logic corresponds exactly to our *agent hypergraph* defined above. Because of space limit and to

focus more on the verification issue, which is the main contribution of this paper, here we only specify the semantics of the argument and commitment operators. The semantics of CTL^{*CA} state formulae is as usual (semantics of CTL^*). A path satisfies a state formula if the initial state in the path does so. Along a path x^i, which starts at state s_i, $\phi_1 \therefore \phi_2$ holds iff ϕ_1 is true and in the next state through the same path if ϕ_1 holds then ϕ_2 holds too. Formally ($\Rightarrow$ stands for material implication):

$$x^i \models_M \phi_1 \therefore \phi_2 \text{ iff } x^i \models_M \phi_1 \text{ and } x^{i+1} \models_M \phi_1 \Rightarrow \phi_2$$

A state s_i satisfies $PC(Ag_1, Ag_2, t, \phi_1)$ iff the commitment is in this state and there is a path along which the commitment content holds. Formally:

$$s_i \models_M PC(Ag_1, Ag_2, t, \phi_1) \text{ iff } PC(Ag_1, Ag_2, t, \phi_1) \in V_{PC}(s_i) \text{ and } s_i \models_M E\phi_1$$

A path x^i satisfies $Act_k(Ag_n, PC(Ag_1, Ag_2, t, \phi_1))$ iff Act_k is in the label of the first transition on this path and in the past[1] $PC(Ag_1, Ag_2, t, \phi_1)$ holds along the same path. Formally:

$$x^i \models_M Act_k(Ag_n, PC(Ag_1, Ag_2, t, \phi_1)) \text{ iff } s_i, Ag_n \xrightarrow{Act_k} s_{i+1} \text{ and } x^i \models_M F^- PC(Ag_1, Ag_2, t, \phi_1)$$

3 Verification

In this section, we will use the $PNAWS$ protocol presented in Section 2.2 to show how our verification approach works. As discussed earlier, we use the MCMAS model checker. In MCMAS, multi-agent systems are described by the Interpreted Systems Programming Language (ISPL), where the system is distinguished into two types of agents: environment agent, which is used to describe boundary conditions and infrastructures, and standard agents. ISPL can also be used to define atomic propositions, action formulae and the specification of properties to be checked. To automatically use this model checker to verify the communication protocol of community of web services, we define a mapping between our *agent hypergraph* and ISPL and encode our CTL^{*CA} formulae in MCMAS.

3.1 Mapping and Transforming Agent Hypergraph to ISPL

In MCMAS, Each agent is composed by: a set of local states, a set of actions, a rule (protocol) describing which action can be performed by an agent, and evolution functions that describe how the local states of the agents evolve based on their current local states and agents' actions [9]. The mapping from our *agent hypergraph* to ISPL is defined by he following rules:

1. $S - to - LocalState$: Every state in the *agent hypergraph* is mapped to the ISPL environment agent, a local state with the same name.
2. $Ag - to - Agent$: Every agent in the *agent hypergraph* is mapped to an ISPL agent with the same name. A special agent environment should also be added to the ISPL agent list.

[1] The past operator F^- is an abreviation and defined as follows: $F^-\phi_1 \equiv trueU^-\phi_1$

3. $V_{PC} - to - LocalValue$: V_{PC} is transformed to local values of the agent that creates the public commitments. The values can be bounded integers, Booleans or enumerations based on the types of these commitments.
4. $Act - to - action/rule$: Every action in the *agent hypergraph* is converted to an ISPL action list of an agent that can execute the action.
5. $\xrightarrow{Act} - to - evolution$: $\xrightarrow{Act}$ is translated into an ISPL agent evolution list. For example if we have $s_i, Ag_n \xrightarrow{Act_k} s_j$, the Ag_n's evolution in ISPL will be:
   `state = s(j) if state = s(i) and Action = Act(k);`
6. $s_0 - to - Init$: s_0 is mapped to an ISPL initial state.

Based on these mapping rules, we use $PNAWS$ as an example to transform the associated *agent hypergraph* into ISPL. In communities of web services, the system includes two types of agents: Master agent and Slave agent. We add an Environment agent to describe the system boundary conditions and infrastructures. Environment agent is a special agent in ISPL system that provides observable variables that can be accessed by other agents. Every agent starts with declaration of local variables. The first mapping rule is used to define the local variables of agent. We declare a state variable to list all possible states in the system:

```
Vars: State: {WaitingforCreate, RefuseReceive...}; end Vars
```

Actions an agents can perform are constructed into Actions section of ISPL file and follow the 4th mapping rule. We also add "null" action to stand for no action. In ISPL Protocol section, we give the permitted actions in each state. Transitions are defined in ISPL Evolution section to show the states change based on the 5th mapping rule.

```
Protocol:
   state = WaitingforCreate: {Create}; ...
end Protocol
Evolution:
   state = ChallengeReceived if state = DefenceSend
   and Slave1.Action=Challenge; ...
end evolution
```

Moreover, we declare a set of initial states. The system starts at state waiting for creating a protocol session with all the counter register reset.

```
InitStates
   (Environment.state = WaitingforCreated) and
   (Master.commitments = toCreate) and
   (Environment.attackCount = 0) and
   (Environment.challengeCount = 0);
end InitStates
```

3.2 Encoding Specifications in ISPL

ISPL specifies both the model and properties. It supports CTL and ATL. Our specifications are expressed by CTL^{*CA}, which extends CTL^*. Therefore, the

basic operators are similar to CTL and we can directly use them in ISPL. For the new operators in CTL^{*CA}, we define rules to convert them into ISPL.

To translate $\phi_1 \therefore \phi_2$ formula into ISPL, we need to declare two variables over the system: `a1` and `a2` to stand for ϕ_1 and ϕ_2. We also need to define the equivalent formula according to the semantics given in Section 2.3, where `X` stands for next and `->` stands for implication.

```
Evaluation
  a1; a2; ...
end Evaluation
Formula a1 and X(a1 -> a2); end Formula
```

For the formulae $PC(Ag_1, Ag_2, t, \phi_1)$ and $Act_k(Ag_n, PC(Ag_1, Ag_2, t, \phi_1))$, the semantics is already encoded in ISPL as V_{PC} and $\xrightarrow{Act}$ are already translated by the 3rd and 5th rules. We just need to define a local value `a1` in agent `Ag1`'s definition to present the commitment, then create the action Act_k over this commitment. The moment `t` is declared in Environment agent because both agents `Ag1` and `Ag2` need to access it at that moment. The code below is an example of the `Create` action (i.e. sending an invitation).

```
Evolution:
   Lvar = Ag2Lvar if moment = t and Ag1.Action = Create and a1;
    ...
end evolution
```

In order to verify the model, we first define some atomic propositions over the system. Thereby, the propositional formulae, which we need to check by MCMAS, are defined based on these propositions.

4 Experimental Results

We have checked different properties of the $PNAWS$ protocol. Here are some examples, where `G` means globally, `F` means in the future and `A` and `E` are the universal and existential quantifiers.

1. Termination: $PNAWS$ always terminates.

   ```
   AG termination
   ```

   `Termination` is an atomic proposition for termination state of the protocol. Intuitively, this property should hold because with finite states and restrained unbound states, a protocol will end.
2. Soundness: The protocol is correct. An example of soundness is: if there is a challenge, a justification will follow in the future.

   ```
   AG (challenge -> F justify)
   ```
3. Reachability: certain states are reachable through any possible sequence of transitions, starting from the initial state. For example, if there is a refusal for an invitation, the protocol will reach defense state.

   ```
   AG (refusal -> F defense)
   ```

4. Liveness: Liveness means something good will eventually happen. An example of liveness is: if there is a negotiation, an acceptance will eventually follow in the future.

```
AG (Attack or Chalenge -> EF Accept)
```

We have implemented the mapping rules and the $PNAWS$ case study scenario along with the specifications in ISPL and verified them with MCMAS. Our system was running on Windows Vista Home Premium on Inter Core 2 Duo CPU T6400 2.00GHz with 3.0GB memory. Experimental results for the example of communities of web services are presented in Table 2. The results clearly show that the state space grows exponentially, but thanks to symbolic model checking the execution time is low.

Table 2. Experimental Results

Number of Slave Agents	1	2	3
Memory in use (MB)	6.6	7.3	9.0
Reachable states	17	393	2,615
Number of BDD Variables	40	60	80
Number of BDD and ADD nodes	6,863	53,690	151,592
Total number of nodes allocated	22,938	83,722	515,756
Execution time (sec)	0.06	0.34	2.04

5 Conclusion and Future Work

Many research proposals [6, 10, 13] have addressed the verification of behavior specifications in UML. Also, extensive research [2, 9, 11, 14, 16] has been done on the verification of multi-agent systems. However, few work focus on these two techniques together. This paper proposes a fully automated approach to verify agent-based communities of web services modeled in UML activity diagrams. We formalized UML activity diagrams using *agent hypergraphs* and specified properties using a new logic for agent communication: CTL^{*CA}. We defined and implemented the mapping rules for transforming the *agent hypergraphs* and CTL^{*CA} specifications into the ISPL language. Finally, we used MCMAS to experiment with the $PNAWS$ protocol.

In this work, action formulae are only captured by the transition labels. Considering the full semantics of different actions in different situations is needed to check more complicated protocols. Our plan for future work is to extend CTL^{*CA} by adding different action formulae and proposing a new OBDD-based algorithm for the model checking. We also plan to extend MCMAS to be fully compatible with this new logic. Besides model checking, we are planning to use Model Driven Architecture (MDA), launched by the Object Management Group (OMG) in 2001 as a promising software design method, to develop a flexible platform for agent-based communities of web services. We intent to use model transformation, which is a process that generates a refined model from a source model [8]. This process is based on a transformation definition, which is a set of transformation rules that describe how one or more constructs in the source language can be transformed into one or more constructs in the target language.

This process can be achieve automatically, which helps in reducing programming errors and coding time.

References

1. Bentahar, J., Maamar, Z., Wan, W., Benslimane, D., Thiran, P. and Subramanian, S. Agent-based communities of web services: An argumentation-driven approach. Service Oriented Computing and Applications, 2(4), 2008, 219-238.
2. Bentahar, J., Meyer, J-J.C. and Wan, W. Model checking communicative agent-based systems. Knowledge-Based Systems, 22(3), 2009, 142-59.
3. Bordini, R. H., Fisher, M., Pardavila, C. and Wooldridge, M. Model Checking AgentSpeak. Proc. of the Int. J. Conf. on Autonomous Agents and Multiagent Systems, ACM, Melbourne, Australia, 2003, 409-416.
4. Cimatti, A., Clarke, E. Giunchiglia, E., Giunchiglia, F., Pistore, M. Roveri, M., Sebastiani, R. and Tacchella, A. NuSMV 2: An open source tool for symbolic model checking. Proc. of the Int. Conf. on Computer Aided Verification, volume 2404 of LNCS, 2002, 359-364.
5. Clarke, E. M., Grumberg, O. and Peled, D. Model checking. MIT Press, Cambridge, Mass., 1999.
6. Eshuis, R. and Wieringa, R. Tool support for verifying UML activity diagrams. IEEE Trans. Software Eng., 30(7), 2004, 437-47.
7. Halpern, J. and Vardi, M.Y. Model checking vs. theorem proving: a manifesto. Proc. of the Int. Conf. on Principles of Knowledge Representation 1991, 325-334.
8. Kleppe, A., Warmer, J. and and Bast, W. MDA Explained, The Model-Driven Architecture Practice and Promise. Addison Wesley, Boston, 2003.
9. Lomuscio, A. Qu, H. and Raimondi, F. MCMAS: A model checker for the verification of multi-agent systems. Proc. of the Int. Conf. on Computer Aided Verification, volume 5643 of LNCS, 2009, 682-688.
10. Planas, E., Cabot, J. and Gomez, C. Verifying action semantics specifications in UML behavioral models. Proc. of the Int. Conf. on Advanced Information Systems Engineering, volume 5565 of LNCS, 2009, 125-140.
11. Raimondi, F. Model Checking Multi-Agent Systems. Ph.D. Thesis, University of London, London, 2006.
12. Rao, A. S. AgentSpeak(L): BDI agents speak out in a logical computable language. Proc. of the European Workshop on Modelling Autonomous Agents in a Multi-Agent World, volume 1038 of LNAI, 1996, 42-55.
13. Schafer, T., Knapp, A. and Merz, S. Model checking UML state machines and collaborations. Elect. Notes in Theoretical Comp. Sc., 55(3), 2001, 357-369.
14. van der Meyden, R. and Gammie, P. MCK: Model checking knowledge. `http://www.cse.unsw.edu.au/~mck/`.
15. Weilkiens, T. System Engineering with SysML/UML Modeling, Analysis, Design. Morgan Kaufmann, Burlington, MA, USA, 2007.
16. Wooldridge, M., Fisher, M., Huget, M. and Parsons, S. Model checking multi-agent systems with MABLE. Proc. of the Int. J. Conf. on Automomous Agents and Multi-Agent Systems. ACM, New York, USA, 2002, 952-959.