Distributed Runtime Verification of JADE Multiagent Systems

Daniela Briola, Viviana Mascardi and Davide Ancona

Abstract Verifying that agent interactions in a multiagent system (MAS) are compliant to a given global protocol is of paramount importance for most systems, and is mandatory for safety-critical applications. Runtime verification requires a proper formalism to express such a protocol, a possibly non intrusive mechanism for capturing agent interactions, and a method for verifying that captured interactions are compliant to the global protocol. Projecting the global protocol onto agents' subsets can improve efficiency and fault tolerance by allowing the distribution of the verification mechanism. Since many real MASs are based on JADE, a well known open source platform for MAS development, we implemented a monitor agent that achieves all the goals above using the "Attribute Global Types" (AGT) formalism for representing protocols. Using our JADE monitor we were able to verify FYPA, an extremely complex industrial MAS currently used by Ansaldo STS for allocating platforms and tracks to trains inside Italian stations, besides the Alternating Bit and the Iterated Contract Net protocols which are well known in the distributed systems and MAS communities. Depending on the monitored MAS, the performances of our monitor are either comparable or slightly worse than those of the JADE Sniffer because of the logging of the verification activities. Reducing the log files dimension, re-implementing the monitor in a way independent from the JADE Sniffer, and heavily exploiting projections are the three directions we are pursuing for improving the monitor's performances, still keeping all its features.

1 Introduction

Verification of the compliance of interaction protocols in distributed and dynamic systems is of paramount importance for most applications. This can

DIBRIS, Genoa University, Italy
e-mail: {daniela.briola,viviana.mascardi,davide.ancona}@unige.it

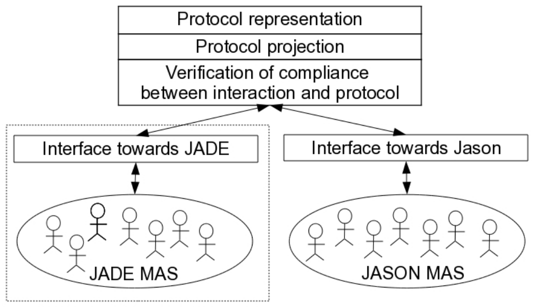


Fig. 1 Our modular framework for distributed runtime verification of MASs.

take place at design-time (offline or static verification) or at runtime (online or dynamic). In the latter case, a layer between the monitor executing the verification and the system under monitoring must exist, so that interactions can be captured and verified against the protocol.

If the system has been engineered as a multiagent system (MAS), then the choice of JADE[1] as the platform for implementing it may be a very natural one. JADE, implemented in Java, is the state-of-the-art tool for MAS development and has been used for many real industrial applications, as described in the JADE Homepage. FYPA (Find Your Path, Agent! [6, 7, 8]) is another industrial MAS developed in JADE and currently being used by Ansaldo STS, the Italian leader in railways signaling and automation, for allocating platforms and tracks to trains inside Italian stations in quasi-real time. Many academic applications spanning different domains are also described in the literature ([4, 13], just to cite a few ones). Due to the wide range of possible application fields and to the large amount of real use cases of JADE, supporting runtime verification of interaction protocols in JADE MASs would be a concrete step towards the reliability reinforcement and the industrial exploitation of MASs: in this paper we describe our contribution for the achievement of this goal.

We have designed and implemented a framework for distributed runtime verification of MASs and a dedicated layer for monitoring JADE interactions. The framework consists of **(1)** a formalism for describing "agent interaction protocols" (AIPs) based on Attributes Global Types (AGT) [1, 10]; **(2)** an algorithm to project AIPs onto subsets of agents, to obtain simpler protocols expressed in the same AGT formalism [2]; **(3)** a mechanism for capturing messages between the JADE agents under monitoring, in a transparent way; and **(4)** a method for verifying that interactions are compliant with the AIP [3].

The strength of our framework, represented in Figure 1, is its high modularity and potential for code reuse, because the first three components are independent from the actual MAS under monitoring. The fourth one (in a

[1] http://jade.tilab.com.

dashed box in the figure) is the subject of this paper, and has been expressly developed for JADE. A layer has been developed for Jason[2] too [3].

The paper is organized as follows: Section 2 describes the design and implementation of the JADE monitor; Section 3 describes the three MASs we have monitored in order to assess the feasibility of our proposal, Section 4 describes our experiments and presents a performance analysis, and Section 5 discusses related approaches and concludes.

2 Runtime Verification of JADE MASs

In order to verify at runtime the interactions taking place in a JADE MAS, we have designed a monitor meeting the following requirements for non intrusiveness and code reuse:

1. the monitor must be able to supervise the execution of the MAS *without interfering with it*,
2. the monitor activity must require *no changes to the agents' code*,
3. the formalism for representing the AIP must be *AGT*,
4. the Prolog code already developed for implementing verification of interactions w.r.t. AGT and for protocol projection *must be re-used as it is.*

To meet requirements 1 and 2 we extended the JADE Sniffer agent, which is able to capture all the messages exchanged during the execution of the MAS in a non intrusive way: JADE is developed under the LGPL (Lesser General Public License) and the Java source code is available to the research community, so we were able to modify it to achieve our goals.

To meet requirements 3 and 4 we exploited the JPL library[3], providing a bidirectional interface between Java and SWI Prolog. As all our previous works on AGT were implemented in Prolog, allowing our JADE Monitor to use Prolog programs and predicates was the best way to ensure reusability.

The monitor is sketched in Figure 2 and is highly modular: we modified the code of the JADE Sniffer's class just as little as possible and we defined the method which converts a JADE message into a Prolog representation amenable for runtime verification in a separate class, to allow developers to modify that class only if a parsing different from the one we provided is required.

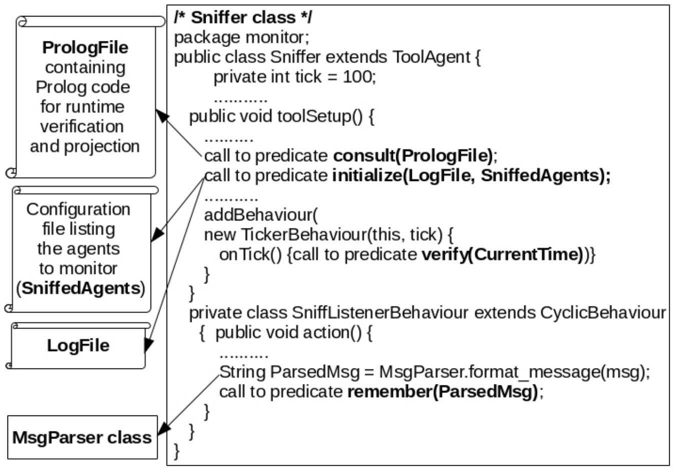
The monitor reads a file containing the Prolog code implementing verification and projection, and a configuration file listing the agents to be monitored, and onto which the protocol projection will be performed. A log file is written as the monitoring goes on.

The Prolog file contains definitions for three predicates:

– `initialize(LogFile, SniffedAgents)`, which sets `LogFile` as the file where writing the outcome of the verification, and projects the global protocol defined by the `global_type/1` predicate onto `SniffedAgents`.

[2] http://jason.sourceforge.net.

[3] http://www.swi-prolog.org/packages/jpl/java_api/.


PrologFile containing Prolog code for runtime verification and projection
Configuration file listing the agents to monitor (SniffedAgents)
LogFile
MsgParser class

```
/* Sniffer class */
package monitor;
public class Sniffer extends ToolAgent {
      private int tick = 100;
      ..........
   public void toolSetup() {
     ..........
     call to predicate consult(PrologFile);
     call to predicate initialize(LogFile, SniffedAgents);
     ..........
     addBehaviour(
     new TickerBehaviour(this, tick) {
        onTick() {call to predicate verify(CurrentTime))}
     }
   }
   private class SniffListenerBehaviour extends CyclicBehaviour
     {  public void action() {
        ..........
        String ParsedMsg = MsgParser.format_message(msg);
        call to predicate remember(ParsedMsg);
        }
     }
}
```

Fig. 2 The JADE monitor.

– `remember(ParsedMsg)`, which inserts the Prolog representation of the JADE captured message into a message list, where messages are ordered by time stamp (if they have a time stamp, which is not mandatory) or in order of arrival.

– `verify(CurrentTime)`, which verifies the compliance to the global protocol of each message remembered in the message list and whose time stamp is lower than `CurrentTime`.

These predicates are called in different methods of the monitor code:

– in the `toolSetup()` method, which initializes the agent, the Prolog file is consulted to make the predicates defined there available, and the `initialize(LogFile, SniffedAgents)` predicate is called;

– in the `action()` method of the `SniffListenerBehaviour` class, the JADE message `msg` is translated into a Prolog term by calling `ParsedMsg = MsgParser.format_message(msg)` and the obtained term is saved into the Prolog message list by calling `remember(ParsedMsg)`;

– a new `Ticker` behavior, re-executed every `tick` milliseconds (`tick` is set to 100 in our setting) is added to the monitor in the setup. This behavior calls the predicate `verify(CurrentTime)`, so that every 100 milliseconds all the messages exchanged in the last 100 milliseconds are verified.

The choice of first remembering the captured messages, and then verifying them, is due to problems with the order in which messages are forwarded to the JADE Sniffer agent, that sometimes do not respect their actual order: if this happens, the monitor could identify a violation of the protocol due to the wrong order of messages when, actually, the violation does not exist. To avoid this risk, we decided to split the interaction verification into two phases. In this way no problems due to the captured messages order arise, provided that the capturing delay is lower than the `tick` value. On the other hand, a violation of the protocol could be identified some milliseconds later, because messages are not checked as soon as they arrive. Our choice of delaying the violation identification rather then raising false violations can be easily changed calling the `verify` predicate as soon as a message is received, after the call to the `remember` predicate. The log file (the excerpt below refers to the Alternating Bit Protocol mentioned in Section 3) stores the result of parsing and verification in the form:

```
Conversion from Jade message (INFORM
  :sender(agent-identifier:name bob@...  :addresses(sequence ...))
  :receiver(set(agent-identifier:name carol@...  :addresses (...)))
  :content  "m2")
to Prolog message
  msg(performative(inform),sender(bob),receiver(carol),content(m2))
which leads from protocol state
(m2:m3:m1:**|(a1, 0):(m1, 1):**)|(m2, 1):(a2, 0):**|(m3, 1):(a3, 0):**
to protocol state
(m3:m1:m2:**|(a1, 0):(m1, 1):**)|(a2, 0):(m2, 1):**|(m3, 1):(a3, 0):**
```

Messages are also printed on the shell, for getting an immediate feedback on the MAS execution.

3 Test Cases

By means of AGT we were able to concisely represent protocols which are well known in the concurrent systems and MAS communities, like the Alternating Bit Protocol (ABP[4]) and the FIPA Iterated Contract Net Protocol (ICNP[5]). We developed two MASs that are expected to adhere to these protocols, in order to verify the ability of our monitor to detect deviations from the expected behavior and to assess its performances.

Our instance of the ABP MAS involves one agent `bob` that sends `m1` to `alice`, `m2` to `carol`, `m3` to `dave`, and waits for their respective acknowledges `a1`, `a2`, `a3` before resending `m1`, `m2`, `m3`, with the constraint that for each iteration i, $\mathtt{m1}_i$ must precede $\mathtt{m2}_i$, which must precede $\mathtt{m3}_i$, and each acknowledge $\mathtt{a}k_i$ must follow $\mathtt{m}k_i$ and precede $\mathtt{m}k_{i+1}$, with k ranging from `1` to `3`.

The ICNP MAS exploits the JADE implementation of the ICNP FIPA protocol offered by the `jade.proto` package[6] and one implementation of the ICNP MAS provided by JADE's developers[7]: in our instance, one `sender` agent playing the role of Initiatior interacts with three `receivers` playing the role of Responder, numbered from `1` to `3`.

The representation of the ABP and ICNP protocols using our AGT formalism is described in [10], where the advantages in terms of readability and conciseness with respect to other existing proposals are widely discussed. Due to space constraints, the reader is invited to refer to [10] for more details.

The FYPA (Find Your Path, Agent!) MAS was developed in JADE starting from 2009. It automatically allocates trains moving into a railway station

[4] en.wikipedia.org/wiki/Alternating_bit_protocol.

[5] fipa.org/specs/fipa00030.

[6] http://jade.tilab.com/doc/api/jade/proto/package-summary.html.

[7] http://jade.tilab.com/doc/examples/protocols.html.

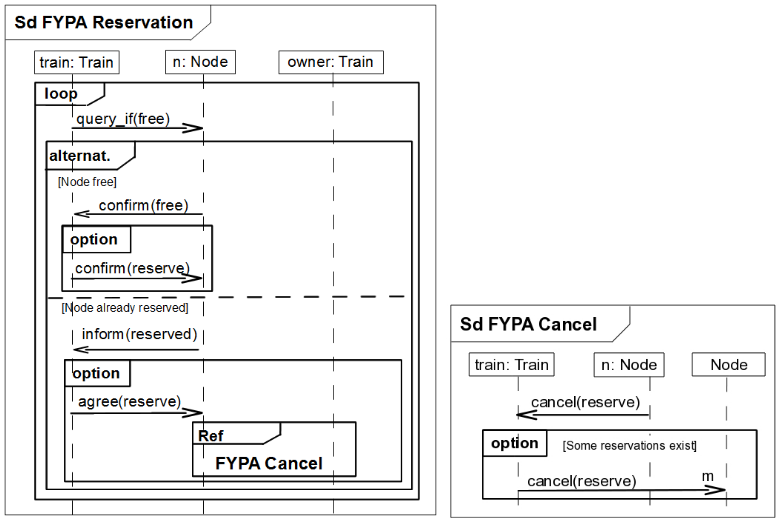


Fig. 3 FYPA Reservation protocol.

to tracks, in order to allow them to enter the station and reach their destination (either the station's exit or a node where they will stop) considering real time information on the traffic inside the station and on availability of tracks. The station can be modeled as a direct non planar graph, where nodes are special railway tracks where trains can stop, and arcs are railway tracks connecting two nodes. The FYPA Reservation protocol described in AUML in Figure 3 involves agents representing trains and nodes. Each train knows the paths $\{P_1 = N_s...N_e; ...; P_k = N_s...N_e\}$ it could follow to go from the node where it is (N_s, for *start*), to the node where it needs to stop (N_e, for *end*). Such paths are computed by a legacy Ansaldo application which is wrapped by an agent named *PathAgent*, not modeled here. Each train also knows which path it is currently trying to reserve, how many nodes answered to its requests and in which way, and how much delay it can accept: to reserve a path, the train must obtain a reservation for each node in it. To reserve a node, a train T_1 asks if it is free, waits for the answer from the node (free or already reserved by another train in an overlapping time slot) and then reserves the resource, which might also mean stealing it to the train T_2 that reserved it before (this usually takes place if the priority of T_1 is higher than that of T_2). In this case, the node will inform train T_2 by following the Cancel protocol, and T_1 will try to reserve the same path in different time slots. Each node knows the arcs that it manages (those that enter in it). It also knows which trains optioned or reserved the node, in which time slots, from which node they are expected to arrive, and which arc they can traverse.

In [11] we presented the formalization of the FYPA protocol using AGTs. That formalization, with minor modifications, has been used for verifying the MAS actual executions as discussed in Section 4.

In the instance of FYPA we tested, train `treno_1` tries to reserve the path `nodo_1, nodo_3, nodo_4, nodo_6` under the following conditions:
– FYPA1: all the nodes in the path are free, as there were no previous reservations: the reservation is completed without any problem;
– FYPA2: there was a previous reservation for one node (`nodo_3`), by a train with priority higher than `treno_1`'s priority: `treno_1` must change the reservation slots for its path;
– FYPA3: there was a previous reservation for one node (`nodo_3`), by a train with priority lower than `treno_1`'s priority: `treno_1` steals the reservation and successfully reserves the full path.

4 Experiments

We tested our JADE monitor with the ABP, the ICNP, and FYPA. With the ABP, which does not use attributes, we were also able to successfully check that projection works as expected. The results were the expected ones in case of both absence and presence of protocol violations.

Because of space constraints, we cannot provide details on all the three MASs. In this section we give the flavor of which kind of properties we were able to test with the FYPA MAS.

The test station consists of six nodes and train `treno_1`, with priority 2, enters from `nodo_1` and then moves to `nodo_3, nodo_4, nodo_6`.

The AGT modeling the FYPA protocol has been described in [11]. As discussed below, we were able to test "local", "horizontal" and "vertical" properties of messages. All our tests gave the expected result, namely a violation was correctly detected when we manually inserted some error in the message content or order, and the protocol verification correctly terminated when we did not insert any error.

"Local" properties of messages. Each message must have the right type. For example, a *query_if* message must be sent by an agent playing the "Train" role, like `treno_1`, to an agent playing the "Node" role, like `nodo_3`, and the arguments of the *query_if* content must contain the priority of the sender, the node from which the train will arrive and a coherent time interval. This message satisfies them:

```
msg(treno_1, nodo_3, query_if, free(2,240000,310000,nodo_1), cid(1), ts(1))
```

"Horizontal" properties of messages sequences. When a train contacts a sequence of nodes to verify whether they are free in order to optionally issue a reservation request, the arguments of the *query_if* messages must form a coherent path: the "From" argument in message m_{i+1} must be the same as the receiver of message m_i, the time slot's first extreme in message m_{i+1} must be the same as the time slot's second extreme in message m_i, the conversation id must be the same, and the train cannot change its priority, apart from setting it equal to infinity (`inf`) for requests than must necessarily be satisfied.

For example, this trace (an extract of a real monitor log file) respects these constraints:

```
msg(treno_1,nodo_1,query_if,free(inf,156000,186000,init), cid(1),ts(1))
msg(treno_1,nodo_3,query_if,free(2,186000,256000,nodo_1), cid(1),ts(2))
msg(treno_1,nodo_4,query_if,free(2,256000,286000,nodo_3), cid(1),ts(3))
msg(treno_1,nodo_6,query_if,free(2,286000,326000,nodo_4), cid(1),ts(4))
```

Note that in the *query_if* message sent to the station entering node (in this case, `nodo_1`), the *From* field is set to the value *initial* (`init`) because there is no "coming from" node (the train is arriving from outside the station).

"Vertical" properties of conversations between a train and a node. Apart from the requirement that during a single conversation the train does not change the conversation id, we can identify one more constraint: if a node is reserved, it must inform the train that sent a *query_if* message of the arc it could have used to reach it and of the time slot when it will be free again. This time slot must start after the time slot's start indicated by the train in its *query_if* message, even if it may overlap with it. A trace like this (again from a real log file) respects both constraints:

```
msg(treno_1,nodo_3,query_if,free(2,24000,31000,nodo_1), cid(1),ts(1))
msg(nodo_3,treno_1,inform,reserved(da0,1,2,dummy,31001,38001),cid(1),ts(2))
```

Since a train can interact with the same node many times, for example because the attempt to reserve a path failed and then the train has to try to reserve a new one, we added and successfully tested another vertical constraint that involves conversation loops: if a train sends more than one *query_if* message to the same node, the conversation id must be different since the messages belong to different conversations.

Performances. Table 1 shows the performance analysis of three categories of execution: with our monitor, with the "plain" JADE Sniffer, with none of them.
– Column **Test** refers to the test we run among those discussed in Section 3.
– Column **R** (for **Runs**) reports the number of runs of a MAS. For example, R equal to 10 means that we performed 10 MAS executions with our monitor, 10 executions with the JADE Sniffer, and 10 executions with none.
– Column **Msg** (for **Messages**) reports the average number of messages exchanged among the agents per run. While in ABP and FYPA the average number is always the same as the exact number per run, as the MAS evolution is deterministic, in the ICNP MAS there is a random choice that participants can make about bidding or not. This means that the runs are not always the same and the number of messages per run can change. We run the MAS many times and we selected 5 runs for each execution category (with monitor, with JADE Sniffer, with none) which show homogeneous features, namely a number of iterations between the initiator and the participants between 4 and 7, and which guarantee that the average number of messages is the same for each category.

– Column **M** (for **Monitor**) reports the average number of milliseconds per message when using our monitor. This value changes from MAS to MAS, as deciding to send one message may require less or more reasoning from the agent, and hence less or more time. **JS** (for **JADE Sniffer**) reports the average number of milliseconds per message when using the JADE Sniffer and **N** (for **None**) reports the average number of milliseconds per message when using none of them.
– Column **M/JS (deg.)** reports the ratio between the performances with our monitor and with the JADE Sniffer and the degradation in percentage ("deg." field in round brackets). Similarly, **M/N (deg.)** reports the performances ratio and degradation between the execution with the monitor and with no JADE built-in agent, and **JS/N (deg.)** reports the performances ratio and degradation between the execution with the JADE Sniffer and with no JADE built-in agent.

Test	**R**	**Msg**	**M**	**JS**	**N**	**M/JS (deg.)**	**M/N (deg.)**	**JS/N (deg.)**
ABP	10	20000	1.93	1.62	0.14	1.19 (19%)	13.78 (1278%)	11.38 (1038%)
ICNP	5	13	12.28	10.47	2.26	1.17 (17%)	5.43 (443%)	4.63 (363%)
FYPA1	5	12	8.10	8.05	2.77	1.01 (1%)	2.92 (192%)	2.90 (190%)
FYPA2	5	20	6.43	6.56	2.63	0.98 (-2%)	2.44 (144%)	2.49 (149%)
FYPA3	5	12	6.61	6.35	2.83	1.04 (4%)	2.33 (130%)	2.24 (124%)

Table 1 Performances of the monitor execution.

For each test, we measured the complete execution time of the MAS. In particular, we measured the number of milliseconds between the start of the protocol (first message sent) and the protocol completion (last message received). Since the ABP is an infinite protocol, we measured the time between `bob`'s setup and the 10000th execution of its `action()` method.

In order to verify the portability of our framework across different operating systems, the experiments with FYPA were run on an Acer 7750 with Intel Core I5 2.3 GHz, 6 GB RAM and Windows 7 Home, whereas the others on an Acer TravelMate 6293 with Intel Core 2 Duo P8400/2.26 GHz, 4 GB RAM, and Mandriva Linux 2009 operating system.

Table 1 shows that the degradation due to the exploitation of the monitor agent with respect to the exploitation of the plain JADE Sniffer is usually between 1% and 19%, with only one test, FYPA2, where the monitor performed slightly better than the JADE Sniffer. The degradation when using the monitor should be mainly due to the fact that the monitor performs many I/O operations for writing the log both on file and on standard output. To make an example, the average dimension of the log files for our ABP tests is 300KB, which justifies the required additional time.

The JADE Sniffer agent is very time-consuming due both to its sniffing capabilities and to its complex graphical interface which requires updates on the fly. Using the JADE Sniffer w.r.t. not using it degrades the MAS performances up to 1038%. It is not surprising then the degradation due to the usage of the monitor w.r.t not using it, up to 1278%, since the monitor adds features to the JADE Sniffer.

From Table 1 we may also notice that the degradation of both the monitor and the JADE Sniffer with respect to using none worsens with the number of exchanged messages. In communication intensive MASs, the presence of agents like the JADE Sniffer and our monitor may represent a bottleneck. By implementing the monitor from scratch instead of relying on the Sniffer agent, keeping the textual interface and removing the GUI, by reducing the dimensions of the monitor's log files reporting only the identified problems, and by exploiting the projections presented in [2] that avoid bottlenecks due to the single centralized monitor, we are confident to overcome most problems related with the monitor's performance.

5 Related Work and Conclusions

Although there are many proposals for runtime verification of agent interaction protocols, that we carefully analyzed in our previous papers on this subject, the attempts to integrate such mechanisms into JADE are, to the best of our knowledge, still missing.

Tools supporting the engineering of JADE MAS are described for example in [12] and [9]. In [12] data mining tools processing the results of the execution of large scale MASs in a monitored environment are discussed. They have been integrated in the INGENIAS Development Kit[8], in order to facilitate the verification of MAS models at the design level rather than at the programming level. The achieved results could be applied to JADE even if, to the best of our understanding, this has not been done. In [9], the authors present a unit testing approach for MASs based on the use of Mock Agents. Each Mock Agent is responsible for testing a single role of an agent under successful and exceptional scenarios. Aspect-oriented techniques are used to monitor and control the execution of asynchronous test cases. The approach has been implemented on top of JADE platform. None of these attempts has the same aim as ours, and thus those proposals and ours cannot be compared. Rather, they could be complemented for providing an integrated framework for engineering and developing JADE MASs.

The work probably most similar to ours, but not interfaced with JADE, is Scribble[9], a tool chain for runtime verification of distributed Java or Python programs against Scribble protocols specifications. Given a Scribble specification of a global protocol, the tool chain validates consistency properties and generates Scribble local protocol specifications for each participant (role) defined in the protocol. At runtime, an independent monitor is assigned to each Java (or Python) endpoint and verifies the local trace of communication actions executed during the session. Besides the different target languages, the main difference of Scribble w.r.t. our work is that we can monitor legacy MASs whose source code is not available because our monitor does not require

[8] ingenias.sourceforge.net/.

[9] http://www.scribble.org.

any change to the agents' code, whereas the Scribble toolchain generates the executable code for the protocol endpoints starting from the specification of the protocol, hence it is suitable for monitoring systems which are created from the protocol specification, but not for legacy ones.

Our implementation of a JADE monitor agent suffers from some limitation, but our tests with three real MASs are very promising. The three problems that we experienced with our monitor are all related to the decision of extending the JADE Sniffer agent. The first is the one described in Section 2, regarding the messages order, the second arises when an agent that is under capturing by the monitor is born: the monitor needs some milliseconds to react and start capturing it, but if in the meanwhile the agent starts sending messages, the monitor could not receive them, and in this case a violation of the protocol is surely identified (even if it is a false positive). The last problem is related with performances, as discussed in Section 4.

We are studying a new version of the monitor that implements a JADE kernel service that captures all messages exchanged by the agents: in this way we should be able to avoid all the three problems above. A comparison with similar solutions including [5] and Scribble is also under way.

References

1. Ancona, D., Barbieri, M., Mascardi, V.: Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In: SAC. ACM (2013)
2. Ancona, D., Briola, D., Seghrouchni, A.E.F., Mascardi, V., Taillibert, P.: Efficient verification of MASs with projections. In: EMAS Pre-proceedings (2014)
3. Ancona, D., Drossopoulou, S., Mascardi, V.: Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In: DALT X, *LNAI*, vol. 7784. Springer (2012)
4. Balachandran, B.M., Enkhsaikhan, M.: Developing multi-agent e-commerce applications with JADE. In: KES (3), LNCS, pp. 941–949. Springer (2007)
5. Baldoni, M., Baroglio, C., Capuzzimati, F.: 2COMM: A commitment-based MAS architecture. In: EMAS, *LNCS*, vol. 8245, pp. 38–57. Springer (2013)
6. Briola, D., Mascardi, V.: Design and implementation of a NetLogo interface for the stand-alone FYPA system. In: WOA, pp. 41–50 (2011)
7. Briola, D., Mascardi, V., Martelli, M.: Intelligent agents that monitor, diagnose and solve problems: Two success stories of industry-university collaboration. In: J. of Inf. Assurance and Security, vol. 4, pp. 106–117 (2009)
8. Briola, D., Mascardi, V., Martelli, M., Caccia, R., Milani, C.: Dynamic resource allocation in a MAS: A case study from the industry. In: WOA (2009)
9. Coelho, R., Kulesza, U., von Staa, A., Lucena, C.: Unit testing in multi-agent systems using mock agents and aspects. In: SELMAS, pp. 83–90. ACM (2006)
10. Mascardi, V., Ancona, D.: Attribute global types for dynamic checking of protocols in logic-based multiagent systems. TPLP **13**(4-5-Online-Supplement) (2013)
11. Mascardi, V., Briola, D., Ancona, D.: On the expressiveness of attribute global types: The formalization of a real multiagent system protocol. In: AI*IA (2013)
12. Serrano, E., Gómez-Sanz, J.J., Botía, J.A., Pavón, J.: Intelligent data analysis applied to debug complex software systems. Neurocomput. **72**(13-15), 2785–2795 (2009)
13. Ughetti, M., Trucco, T., Gotta, D.: Development of agent-based, peer-to-peer mobile applications on ANDROID with JADE. In: UBICOMM (2008)