A Framework for Monitoring the Execution of Distributed Multi-agent Programs

Eslam Al Maghayreh, Samer Samarah, Faisal Alkhateeb, Iyad Abu Doush,
Izzat Alsmadi and Ahmad Saifan
Faculty of Information Technology and Computer Science
Yarmouk University
Irbid 21163, Jordan.
{*eslam, samers, alkhateebf, iyad.doush, ialsmadi, ahmads*}*@yu.edu.jo*

Abstract

Developing dependable distributed multi-agent programs is not an easy task. Even with extensive testing and debugging, faults cannot be completely removed. However, many distributed multi-agent programs, especially those employed in safety critical environments, should be able to function properly even in the presence of software faults. Monitoring the execution of a distributed multi-agent program, detecting failures, and reacting to these failures at runtime is the appropriate way to tolerate these failures.

In this paper, we have designed and implemented a framework for monitoring the execution of distributed multi-agent programs. The proposed framework extends the ***J****ava* ***A****gent* ***DE****velopment framework (JADE) allowing agent programmers to monitor global states, to detect the occurrence of certain events and to react to these events at runtime. An example of monitoring a distributed multi-agent program has been presented to demonstrate the effectiveness of the proposed framework.*

Keywords**:** *Distributed systems, Monitoring, Multi-agent programs, Distributed predicates detection, Java Agent DEvelopment framework (JADE), Testing, Debugging, Runtime verification.*

1. Introduction

Multi-agent systems have been brought up with the research and development of distributed systems. Many new multi-agent applications are currently being developed and launched [1, 2, 3]. Developing distributed multi-agent applications is a difficult task [4, 5]. Programmers need to put more effort to take care of additional issues like communication, synchronization, problems of deadlock, etc.

In fact, writing dependable distributed multi-agent programs is very hard. Even if extensive testing and debugging mechanisms are applied, faults persist. Many distributed multi-agent systems should be able to function properly even in the presence of software faults. Monitoring the execution of a distributed multi-agent program and detecting failures is an important

technique to tolerate such faults. This gives rise to the monitoring and runtime verification techniques [6, 7, 8, 9, 10, 11].

Testing and formal methods, such as model checking and theorem proving, can be used to detect bugs in distributed multi-agent programs. However, in traditional testing, we cannot formally specify the properties that a program needs to satisfy. Formal methods are usually applied on an abstract model of the program. Consequently, even if a program has been formally verified, we cannot guarantee the dependability of a particular implementation.

The idea of verifying whether a run of a distributed program satisfies a given property (runtime verification) has been recently attracting much attention in the research area of verifying the correctness of distributed programs [5, 11, 12, 13, 14, 15]. Many problems in distributed programs can be reduced to the problem of monitoring a run of a distributed program to check whether it satisfies a given property or not. Termination detection and deadlock detection are some examples [5].

In this paper, we have designed and implemented a framework to monitor the execution of a distributed multi-agent program, report the occurrence of certain events (formally represented as predicates) and react to these events accordingly at runtime. We have provided an interface for implementing a monitored agent to expose local states to be observed, and we have implemented an algorithm to detect a well known class of predicates known as Weak Conjunctive Predicates (WCP) [5]. The proposed framework has been developed on a widely used agent platform (JADE) and its functioning has been demonstrated through an example.

The rest of this paper is organized as follows: Section 2 presents a formal model of a run of a distributed multi-agent program. The main difficulties of distributed predicates detection (also referred to as runtime verification) are discussed in Section 3. Section 4 presents the main distributed predicate detection approaches. An algorithm to detect weak conjunctive predicates is illustrated in Section 5. An overview of the Java Agent DEvelopment framework (JADE) is presented in Section 6. After that, the architecture of the proposed framework is described in detail in Section 7. Some details about the implementation of the framework along with an example to demonstrate its effectiveness are presented in Section 8. Finally, a summary of the main conclusions and possible future works are presented in Section 9.

2. A formal model of a run of a distributed multi-agent program

We assume a message-passing distributed multi-agent program consisting of n agents denoted by $A_1, A_2, ..., A_n$ and a set of unidirectional channels. The state of a channel at any time is represented by the sequence of messages sent along that channel but not yet received.

Definition 1 *An* **event** *is the result of executing a statement in a program. It can be a computational event or a communication event (send/recieve).*

events are related by either their execution order in an agent or message send/receive relations among agents. The happened-before relation ($\rightarrow$) defined by lamport can be applied to all events executed [16].

Definition 2 *A* **run** *of a distributed multi-agent program is formally represented as an event structure $\langle E, \rightarrow \rangle$ consists of the set of events executed (E) and the happened-before relation $(\rightarrow)$ among these events.*

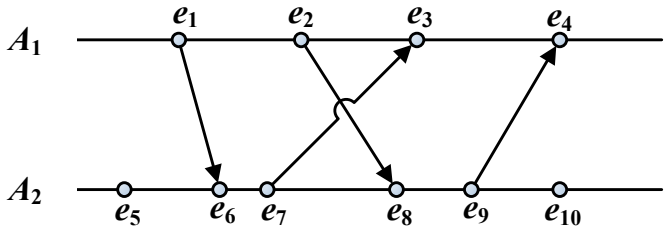

Figure 1. A space-time diagram representing a run of a distributed multi-agent program involving two agents (A_1 and A_2).

A two dimensional space-time diagram can be used to represent a run of a distributed multi-agent program. The horizontal direction represents time and the vertical direction represents space. Events are represented by circles. Each send event is connected with the corresponding receive event using a directed edge. The space-time diagram shown in Figure 1 represents a run of a distributed multi-agent program involving two agents.

The events of a single agent are totally ordered. An event e_1 is happened before another event e_2 (denoted by $e_1 \rightarrow e_2$) if and only if there is a directed path from e_1 to e_2. The $\rightarrow$ relation is a partial order relation. Two events e_1 and e_2 are said to be ***concurrent*** (denoted by $e_1 \parallel e_2$) if they are not related by the $\rightarrow$ relation. For example, in Figure 1, $e_1 \parallel e_5$ because $\neg(e_1 \rightarrow e_5)$ and $\neg(e_5 \rightarrow e_1)$.

Definition 3 *A* **consistent cut** X *of a run of a distributed multi-agent program* $\langle E, \rightarrow \rangle$ *is a finite subset* $X \subseteq E$ *such that if* $a \in X \ \wedge \ b \rightarrow a$ *then* $b \in X$.

A ***global state*** of a run is represented by the values of the program variables and channels states attained upon completion of the events in a given consistent cut X. A ***state lattice*** is the set of global states of a given run endowed with set union and set intersection operations [17]. Figure 2 shows the state lattice of the run shown in Figure 1. $(-,-)$ is the initial state. (e_2, e_7) is the state reached after executing event e_2 in A_1 and event e_7 in A_2. (e_4, e_{10}) is the final state. Based on the state lattice, we can verify whether a run of a given distributed multi-agent program satisfies certain properties or not (runtime verification).

In the following section, we will present the main reasons that make monitoring and predicates detection a difficult task in distributed multi-agent programs.

3. Difficulties of predicates detection in distributed multi-agent programs

Monitoring a run of a distributed multi-agent program to check if it satisfies a given property (represented as a predicate) is a difficult task. The difficulty comes from the characteristics of a distributed multi-agent program that can be summarized as follows [5]:

1. There is no common clock between agents. Consequently, the events in a given run of a distributed multi-agent program can only be partially ordered.
2. There is no shared memory between the agents. Consequently, there will be message overhead to collect the information necessary to evaluate a predicate.

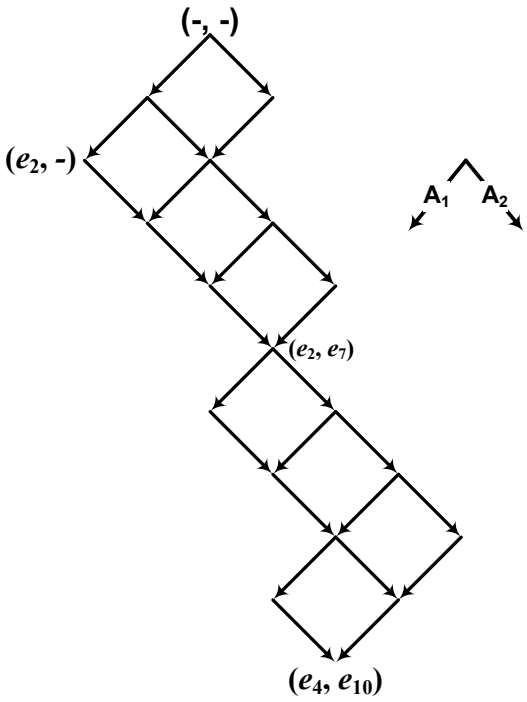

Figure 2. The state lattice of the run shown in Figure 1.

3. There are multiple agents running concurrently. Consequently, the number of global states that must be considered in evaluating a predicate will be exponential in number of agents.

In the next section we will briefly explore the approaches presented in the literature to check the satisfaction of a predicate in a run of a given distributed program.

4. Approaches of predicates detection in distributed programs

There are three main approaches to check the satisfaction of a given predicate in a run of a distributed program. The first approach is based on the global snapshot algorithm proposed by Chandy and Lamport [18, 19, 20]. According to this approach, a global state of a run will be captured and the desired predicate will be evaluated on it, if the predicate is not evaluated to true, another global state will be captured. This process will continue until a global state is found in which the desired predicate is satisfied. This approach works well for predicates that do not turn false once they become true (stable predicates).

The second approach to detect the satisfaction of a predicate in a run of a given distributed program was proposed by Cooper and Marzullo [21]. This approach is based on the construction of the state lattice of the run under consideration. It can be used to detect $possibly : \phi$ and $definitely : \phi$. $possibly : \phi$ is true if the predicate ϕ is evaluate to true in at least one global state in the state lattice. $definitely : \phi$ is true if, for all paths from the initial global state to the final global state, ϕ is true in at least one global state along that path [21, 22]. This approach can be used to detect both stable and unstable predicates. However, the detection is

very expensive because it requires exploring $O(m^n)$ global states in the worst case, where n is the number of processes (agents) and m is the number of local states in each process.

The third approach exploits the structure of the predicate to avoid the construction of the state lattice. According to this approach, a subset of the global states is identified based on the structure of the predicate such that if the predicate is true, it must be true in one of the states in this subset. This approach is less general than the second approach but it can be used to develop more efficient algorithms for certain classes of predicates. In [23, 24], Garg and Waldecker have presented algorithms of complexity $O(n^2m)$ to detect $possibly : \phi$ and $definitely : \phi$ when ϕ is a conjunction of local predicates (a local predicates is a predicates that can be expressed in terms of the variables of a single process (or agent)).

In general, the problem of detecting a global property in a run of a given distributed program is NP-complete [13]. Consequently, the class of the predicate to be detected must be restricted to allow for efficient detection. In this paper, we will deal with local predicates and global predicates that can be represented as a conjunction or as a disjunction of local predicates. It has been proved in the literature that these classes of predicates can be efficiently detected [5].

A conjunctive predicate P under the modality $possibly$ is called a *Weak Conjunctive Predicate* (WCP) if all the terms of the predicate are local predicates [5]. A weak conjunctive predicate is true for a given run if and only if there is a global state in that run in which all the local predicates involved in it are evaluated to true. Detecting the satisfaction of WCPs is generally useful in detecting a combination of states that is unsafe [5].

5. Detecting weak conjunctive predicates

In [5, 23], an efficient algorithm to detect WCPs has been presented. The algorithm finds the first consistent cut for which a WCP is true. In this algorithm, one agent will serve as a monitor. All other agents involved in detecting the WCP are referred to as application agents.

Each term in any WCP is a local predicate associated with exactly one of the application agents. It is the responsibility of each application agent to monitor its local predicate and to maintain the vector clock algorithm. The vector clock is a very well known technique to assign timestamps to the events of a run of a given distributed program [17, 25]. When the local predicate becomes true for the first time since the most recently sent message (or the beginning of the execution), the application agent sends to the monitor a monitoring message containing its local state (in this algorithm the local state of an agent consist of the timestamp vector of the event that makes the agent's local predicate evaluates to true).

The monitor receives monitoring messages from the application agents that are involved in detecting the WCP. It is responsible for searching a global state (consistent cut) that satisfies the WCP. The monitor examines a sequence of candidate cuts, if the candidate cut is not a consistent cut or does not satisfy all the terms of the WCP, the monitor will remove one of the local states along the cut. The eliminated local state cannot be part of any consistent cut that satisfies the WCP. The monitor then moves to the next candidate cut by considering the successor to the eliminated local state on the cut. If the monitor finds a consistent cut for which no local state can be eliminated, then that cut satisfies the WCP and the detection algorithm halts. More details about this algorithm can be found in [5, 23].

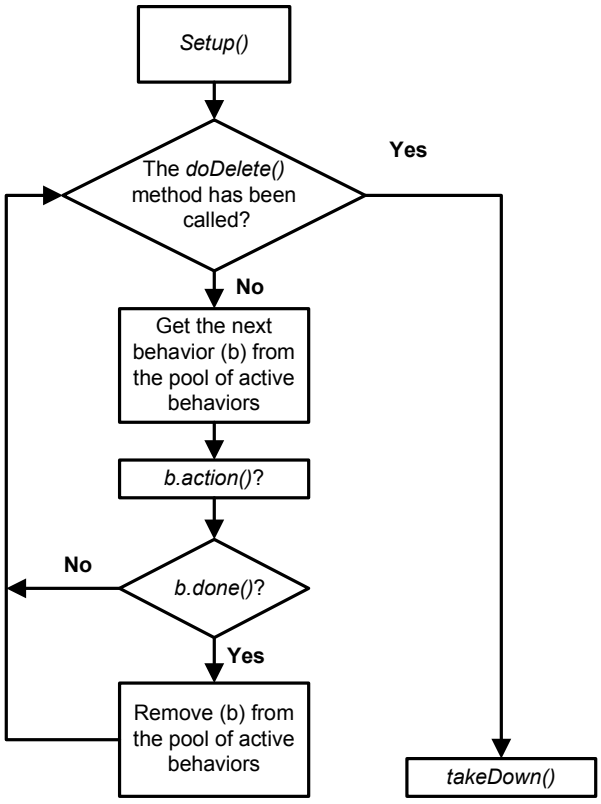

Figure 3. The execution path of an agent in JADE [28].

6. JADE: Java Agent DEvelopment framework

JADE is a framework for developing multi-agent applications [26]. It is provided as a FIPA-compliant agent framework with a package to develop Java agents [27]. JADE is fully implemented in Java and it is totally open source.

In JADE framework, there is an agent container in each host to hold its local agents. Each agent is an active thread that has its own behaviors. Agents in JADE are executed concurrently. JADE provides a virtual agent platform, by which all agents can interact with each other, regardless of their containers or hosts.

The execution path of an agent is shown in Figure 3. The *setup()* method is invoked by the JADE runtime once an agent starts and it is intended to include agent initializations. The *takeDown()* method is invoked just before an agent terminates and is intended to include agent clean-up operations [28].

The actual task an agent has to perform is typically carried out within "behaviors". An agent can execute several behaviors at the same time. However, it is important to know that scheduling of behaviors in an agent is cooperative (not preemptive). Consequently, when a behavior is scheduled for execution its *action()* method is invoked and runs until it returns. Therefore, it is the programmer who decides when an agent switches from the execution of a behavior to the execution of another one [28]. More information about JADE can be found at [26].

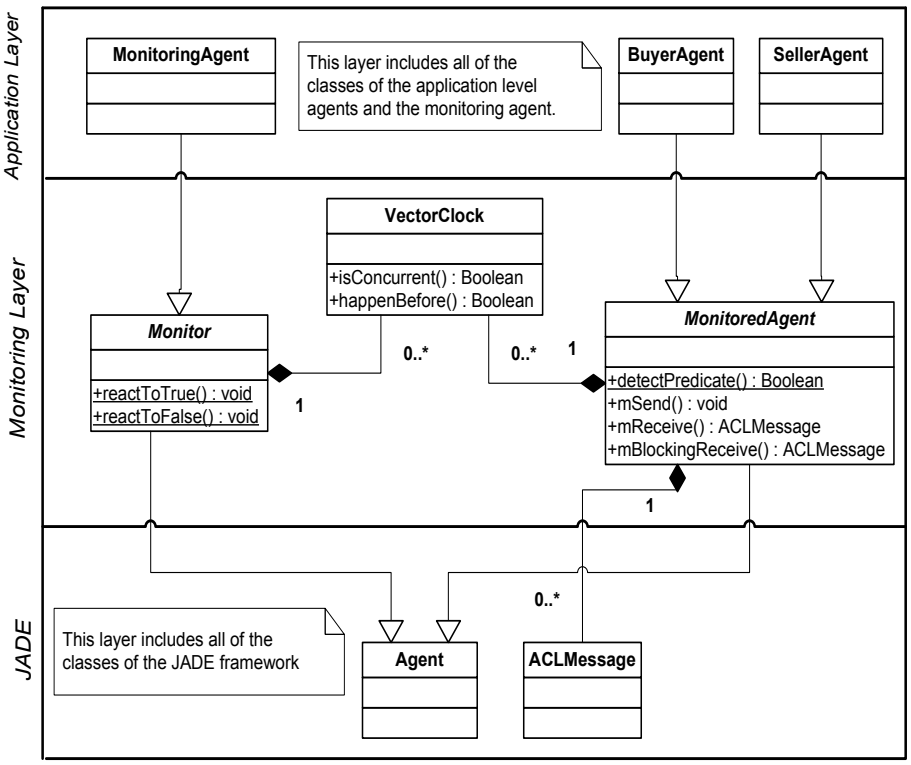

Figure 4. The architecture of the proposed monitoring framework.

7. The architecture of the proposed framework

The architecture of the proposed monitoring framework consists of three layers as shown in Figure 4. The bottom layer is the JADE framework, the middle layer is the monitoring layer, and the top layer is the application layer.

The layered architecture indicates that the internal implementation of any layer can be changed without affecting other layers as long as the interface between these layers remains the same. Each layer provides functions to the layer above it. The interface between the layers includes the necessary information that tells a layer how to use the functions provided by the layer below it.

The monitoring layer contains all the necessary functions to monitor the execution of a given distributed multi-agent program and to report the occurrence of certain events of interest. It contains three main classes, *Monitor*, *MonitoredAgent* and *VectorClock*. The *Monitor* class extends the *Agent* class of the JADE framework and defines all the necessary functions to be used by a monitoring agent to collect the necessary information at runtime and to report the occurrence of certain events that are represented as predicates. The class *Monitor* involves two abstract methods, *reactToTrue()* and *reactToFalse()*. These two methods are defined as abstract methods because different applications may react in different ways for the same event, so the implementations of these two methods have been left for the application developer in the application layer.

The *MonitoredAgent* class extends the *Agent* class of the JADE framework. All the agents at the application level should be objects of classes that extend the *MonitoredAgent* class.

For example, the *SellerAgent* class in Figure 4 extends the *MonitoredAgent* class. The *MonitoredAgent* class redefines the communication operations (send, receive, blockingRecevie) in such a way that allows us to assign timestamps to all of the communication events. The new version of the communication operations are called *mSend()*, stand for monitored send operation, *mRecevie()* and *mBlockingRecevie()*. The *mSend()* operation adds a timestamp to the message to be sent and then invokes the original *send()* operation, whereas the *mRecevie()* and *mBlockingRecevie()* operations receive a message using the original *recevie()* and *blockingRecevie()* operations and then remove the timestamp from the received message before delivering it.

The *MonitoredAgent* class involves an abstract method called *detectPredicate()* to be implemented by the application developer, this method represents the local predicate to be monitored locally by each agent.

The *VectorClock* class includes all the necessary functions to assign timestamps to the events relevant to the global properties/predicates to be monitored.

According to the specification of the architecture of the framework described above, the application developer will not build his application directly on top of the JADE framework, but he will build it on top of our monitoring layer. For example, when he wants to define a class of agents he will not extend the *Agent* class in JADE, but he will extend the *MonitoredAgent* class in the monitoring layer of our framework.

In the following section, we will demonstrate the functionality of the proposed framework through an example.

8. Experimental results

In this section, we will give a brief description of a distributed multi-agent application built on top of our monitoring layer. The application is a simple e-market application where there are a number of seller agents and a number of buyer agents. The seller agents can sell different kinds of products where the price of each product may change from time to time. When a buyer agent wants to buy a product, he will send a call for proposal for each seller agent selling the product of interest. After receiving the proposals from the seller agents, he will buy the product from the seller agent who has proposed the lowest price.

To build this application, we have defined two classes *SellerAgent* and *BuyerAgent*, both of them extend the *MonitoredAgent* class (see Figure 4). The two classes can be used to create as many seller and buyer agents as we want.

Suppose we want to monitor the price of a product, say x, to make sure that not all of the seller agents sell this product at a price greater than 13$. To do so, we have to create a monitoring agent (an agent of the class *MonitoredAgent*). The monitoring agent will look for the agents selling the desired product and monitor them. This problem can be reduced to detecting whether the following conjunctive predicate has been detected at some point in time during the execution of the e-market application.

$$(p_1 > 13) \wedge (p_2 > 13) \wedge \cdots \wedge (p_n > 13)$$

Where n is the number of seller agents selling product x, and p_i is the price of product x offered by seller agent i.

This conjunctive predicate consists of n terms $(p_1 > 13), (p_2 > 13), \ldots, (p_n > 13)$. Each term is called a local predicted (means that the value of the predicate can be evaluated locally by a single agent). The above conjunctive predicate is called a global predicate (means that the monitoring agent can not evaluate it locally without collecting the necessary information from all of the agents involved in the predicate).

In our example, the monitoring agent will keep monitoring the agents of interest (agents selling product x) to collect all of the information that may affect the value of the above conjunctive predicate. Once the predicate is detected, the monitoring agent will react accordingly (through calling the *reactToTrue()* method).

The reaction to the satisfaction of a given predicate is application dependent. For example, in some applications, when the above predicate is satisfied, the monitoring agent may send a message to the seller agents asking them to reduce the price of the product. In other applications, the satisfaction of a similar predicate may indicate that there is some bug in the system that has to be removed. This is why the *reactToTrue()* method has been defined as an abstract method in the monitoring layer and the actual implementation has been left to the application layer.

In our demonstration example, the *reactToTrue()* method will just display a message indicating that the predicate has been satisfied along with the timestamps of each local state involved in the global state that satisfies the predicate.

The proposed framework has been used to monitor the above mentioned predicate and to report its satisfaction. Part of the detailed output displayed by the seller agents and the monitoring agent are as follow:

```
Seller Agent A@107PC:1099/JADE Start Updating the Catalog
My ID is: 285100506
My local predicate is: p > 13

Seller Agent B@107PC:1099/JADE Start Updating the Catalog
My ID is: 1638410203
My local predicate is: p > 13

Seller Agent C@107PC:1099/JADE Start Updating the Catalog
My ID is: 1303247396
My local predicate is: p > 13

B Sends a message to C
Agent Name: B, Current Price = 5
Vector Clock = 1638410203 [(1638410203:1)]
C Receives a message from B
Agent Name: C, Current Price = 5
Vector Clock = 1303247396 [(1303247396:1)(1638410203:1)]
B Sends a message to A
Agent Name: B, Current Price = 10
Vector Clock = 1638410203 [(1638410203:3)]
A Receives a message from B
Agent Name: A, Current Price = 5
```

Vector Clock = 285100506 [(285100506:1)(1638410203:3)]

MonitoringAgent M@107PC:1099/JADE is ready.
The Monitoring Agent is looking for the agents to be monitored...
Agent found:(agent-identifier :name B@107PC:1099/JADE :addresses (sequence http://107PC:7778/acc)) Selling: x
Agent found:(agent-identifier :name C@107PC:1099/JADE :addresses (sequence http://107PC:7778/acc)) Selling: x
Agent found:(agent-identifier :name A@107PC:1099/JADE :addresses (sequence http://107PC:7778/acc)) Selling: x
The Monitoring Agent has found All the agents to be monitored.

A Sends a message to C
Agent Name: A, Current Price = 10
Vector Clock = 285100506 [(285100506:3)(1638410203:3)]
C Receives a message from A
Agent Name: C, Current Price = 10
Vector Clock = 1303247396 [(1303247396:3)(285100506:3)(1638410203:3)]
B Sends a message to C
Agent Name: B, Current Price = 15
Vector Clock = 1638410203 [(1638410203:5)]
C Receives a message from B
Agent Name: C, Current Price = 15
Vector Clock = 1303247396 [(1303247396:5)(285100506:3)(1638410203:5)]
B Sends a message to A
Agent Name: B, Current Price = 20
Vector Clock = 1638410203 [(1638410203:7)]
A Receives a message from B
Agent Name: A, Current Price = 15
Vector Clock = 285100506 [(285100506:5)(1638410203:7)]

The Predicate: $B \wedge C \wedge A$ has been detected at cut:
B: 1638410203 [(1638410203:8)]
C: 1303247396 [(1303247396:6)(285100506:3)(1638410203:5)]
A: 285100506 [(285100506:6)(1638410203:7)]

A Sends a message to C
Agent Name: A, Current Price = 20
Vector Clock = 285100506 [(285100506:7)(1638410203:7)]
C Receives a message from A
Agent Name: C, Current Price = 20
Vector Clock = 1303247396 [(1303247396:7)(285100506:7)(1638410203:7)]
B Sends a message to C
Agent Name: B, Current Price = 25
Vector Clock = 1638410203 [(1638410203:9)]

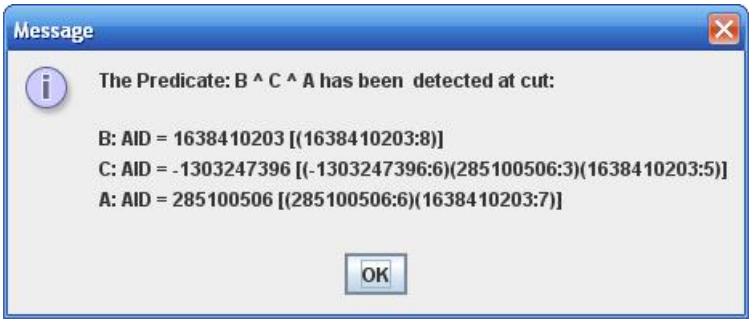

Figure 5. The message displayed by the monitoring agent indicating that the predicate to be monitored has been detected.

In this example, we have created three seller agents A, B and C. Each of them is selling product x. Line number one in the above output is the first statement printed by the seller agent A. It indicates that the agent is in the process of adding the information about product x to his catalogue (a data structure where all of the necessary information about all of the products available for sale are maintained). After that, the agent prints its unique identification number (ID) and the local predicate that it will be monitoring.

The statement in line 13 indicates that the seller agent B has sent a message to the seller agent C informing him that he has changed the price of product x to 5$ as indicated in line 14.

Line 26 is the first statement printed by the monitoring agent M; it indicates that the monitoring agent M has just started. The monitoring agent will automatically look for the agents selling the product of interest. In line 34, the monitoring agent reports the fact that he has found all of the agents selling product x and they are all monitored by him starting from this point of time.

In line 55, the monitoring agent M has detected the predicate of interest and printed a detailed message indicating the agents involved in the predicate and the vector clock timestamp of each local state involved in the global state that satisfies the predicate. For simplicity, the predicate is written using agents' names instead of the local predicates. Figure 5 shows the message displayed by the monitoring agent as a response to the detection of the predicate.

9. Conclusion and future work

In this paper, we have designed and implemented a framework for monitoring the execution of distributed multi-agent programs, detecting the occurrence of certain events of interest and reacting to these events accordingly. Monitoring can provide the user with additional information about the program's run-time behavior. This information cannot be obtained by analyzing the program's source code alone. The satisfaction of a predicate of interest may indicate that there is some bug in the system or that there is a situation where direct reaction at runtime is required. Consequently, the use of the proposed framework will help programmers in developing more dependable distributed multi-agent applications.

The size of the information that a monitor has to receive and process in order to detect the satisfaction of a certain predicate can be very large, especially in a distributed environment. This work can be extended by supplying the monitoring layer with additional mechanisms to

reduce the amount of data to be processed by the monitor. The notion of atomicity can be exploited in this regard. Viewing distributed multi-agent programs as programs that execute a set of atomic actions can significantly simplify the monitoring and runtime verification of these programs [4, 29].

References

[1] F. Alkhateeb, E. Al Maghayreh, and I. Abu Doush, *Multi-Agent Systems - Modeling, Control, Programming, Simulations and Applications*, InTech, 2011.

[2] F. Alkhateeb, E. Al Maghayreh, and I. Abu Doush, *Multi-Agent Systems - Modeling, Interactions, Simulations and Case Studies*, InTech, 2011.

[3] Piotr Jedrzejowicz, Ngoc Thanh Nguyen, Robert J. Howlett, and Lakhmi C. Jain, Eds., *Agent and Multi-Agent Systems: Technologies and Applications, 4th KES International Symposium, KES-AMSTA, Gdynia, Poland*, vol. 6070 of *Lecture Notes in Computer Science*. Springer, 2010.

[4] E. Al Maghayreh, *Simplifying Runtime Verification of Distributed Programs: Ameliorating the State Space Explosion Problem*, VDM Verlag, 2010.

[5] Vijay K. Garg, *Elements of distributed computing*, John Wiley & Sons, Inc., New York, NY, USA, 2002.

[6] Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, Eds., *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta*, vol. 6418 of *Lecture Notes in Computer Science*. Springer, 2010.

[7] Saddek Bensalem and Doron Peled, Eds., *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France*, vol. 5779 of *Lecture Notes in Computer Science*. Springer, 2009.

[8] Martin Leucker, Ed., *Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary*, vol. 5289 of *Lecture Notes in Computer Science*. Springer, 2008.

[9] Oleg Sokolsky and Serdar Tasiran, Eds., *Runtime Verification, 7th International Workshop, RV 2007, Vancover, Canada*, vol. 4839 of *Lecture Notes in Computer Science*. Springer, 2007.

[10] Tingting Hua, Yu Huang 0002, Jiannong Cao, and XianPing Tao, "A lattice-theoretic approach to runtime property detection for pervasive context," in *the 7th International Conference on Ubiquitous Intelligence and Computing*, 2010, pp. 307–321.

[11] Özalp Babaoğlu, Eddy Fromentin, and Michel Raynal, "A unified framework for the specification and runtime detection of dynamic properties in distributed computations," *J. Syst. Softw.*, vol. 33, no. 3, pp. 287–298, 1996.

[12] Craig M. Chase and Vijay K. Garg, "Efficient detection of restricted classes of global predicates," in *WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms*. 1995, pp. 303–317, Springer-Verlag.

[13] Craig M. Chase and Vijay K. Garg, "Detection of global predicates: Techniques and their limitations," *Distributed Computing*, vol. 11, no. 4, pp. 191–201, 1998.

[14] Patrice Godefroid, Michael Y. Levin, and David Molnar, "Active Property Checking," Tech. Rep. MSR-TR-2007-91, Microsoft Research, 2007.

[15] Usa Sammapun, Insup Lee, and Oleg Sokolsky, "RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties," in *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2005, pp. 147–153.

[16] Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[17] Friedemann Mattern, "Virtual Time and Global States of Distributed Systems," in *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Château de Bonas, France, October 1989, pp. 215–226.

[18] K. Mani Chandy and Leslie Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[19] L. Bougé, "Repeated snapshots in distributed systems with synchronous communications and their implementation in CSP," *Theor. Comput. Sci.*, vol. 49, pp. 145–169, January 1987.

[20] Madalene Spezialetti and Phil Kearns, "Efficient distributed snapshots," in *Proceedings of the 6th International Conference on Distributed Computing Systems*, Massachusetts, USA, 1986, pp. 382–388.

[21] Robert Cooper and Keith Marzullo, "Consistent detection of global predicates," *SIGPLAN Not.*, vol. 26, no. 12, pp. 167–174, 1991.

[22] Roland Jegou, Raoul Medina, and Lhouari Nourine, "Linear space algorithm for on-line detection of global predicates.," in *Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT), Berlin*, 1995, pp. 175–189.

[23] Vijay K. Garg and Brian Waldecker, "Detection of weak unstable predicates in distributed programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 3, pp. 299–307, 1994.

[24] Vijay K. Garg and Brian Waldecker, "Detection of strong unstable predicates in distributed programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 12, pp. 1323–1333, 1996.

[25] Colin Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," in *Proceedings of the 11th Australian Computer Science Conference* , 1988, pp. 56–66.

[26] JADE: Java Agent DEvelopment Framework, "http://jade.tilab.com/," accessed October 1, 2011.

[27] FIPA: the Foundation for Intelligent Physical Agents, "http://www.fipa.org," accessed October 1, 2011.

[28] Giovanni Caire, "Jade tutorial: Jade programming for beginners," *TILAB*, 2009.

[29] L. Lamport, "A theorem on atomicity in distributed algorithms," *Distributed Computing*, vol. 4, no. 2, pp. 59–68, 1990.

Authors

Eslam Al Maghayreh is an Assistant Professor in the Department of Computer Science at Yarmouk University (Jordan) since June 2008. He has received his PhD degree in Computer Science from Concordia University (Canada) in 2008, his M.Sc. degree in Computer Science from Yarmouk University in 2003, and his B.Sc. degree in Computer Science from Yarmouk University in 2001. His research interests include: Distributed systems, multi-agent systems, and runtime verification of distributed programs.

Samer Samarah received the Ph.D. degree in computer science from the University of Ottawa, Ottawa, ON, Canada, in 2008. He is currently an Assistant Professor with the Department of Computer Information System, Yarmouk University, Irbid, Jordan, and a Research Associate with the School of Information Technology and Engineering, University of Ottawa. His research interests are Software Engineering, wireless networks, wireless ad hoc and sensor networks, and data mining for both distributed systems and wireless sensor networks.

Faisal Alkhateeb is an assistant professor in the department of computer science at Yarmouk University. He holds a B.Sc. from Yarmouk University, 1999; a M.Sc. from Yarmouk University, 2003; a M.Sc. from Grenoble 1, 2004; and Ph.D. from Grenoble 1, 2008. He is interested in knowledge-based systems, knowledge representation and reasoning, constraint satisfaction and optimization problems, intelligent systems, Semantic Web, and Artificial Intelligence.

Iyad Abu Doush is an Assistant Professor in the Department of Computer Science at Yarmouk University, Jordan. Dr. Abu Doush has received his BSc. In Computer Science from Yarmouk University, Jordan, 2001, and his M.S. from Yarmouk University, Jordan, 2003. He earned his Ph.D. from the Computer Science Department at New Mexico State University, USA, 2009. Since then, he has been a professor of computer science, Yarmouk University, Jordan. He has published more than 20 articles in international journals and conferences. His research interests include assistive technology, intelligent interfaces, and multimodal interfaces. Other research interests include human computer interaction, computational intelligence, and collaborative virtual environments.

Izzat Alsmadi is an assistant professor in the department of computer information systems at Yarmouk University in Jordan. He obtained his Ph.D degree in software engineering from NDSU (USA). His second master in software engineering from NDSU (USA) and his first master in CIS from University of Phoenix (USA). He had B.sc degree in telecommunication engineering from Mutah university in Jordan. Before joining Yarmouk University he worked for several years in several companies and institutions in Jordan, USA and UAE. He has several published books: Advanced Automated Software Testing: Frameworks for Refined Practice(2011-2012), Building a GUI Test Automation Framework Using the Data Model (2008) , Software metrics, toward building proxy models, (2008) , Software Engineering Ontology Separation of Concerns (2009).

Ahmad Saifan is an assistant professor in the department of computer information systems at Yarmouk University (YU) in Jordan. He obtained his Ph.D degree in software engineering from Queen's University (Canada). His master degree is in Computer Science from YU. He had B.Sc degree in computer science from YU. His research interests include: Software testing, debugging and verification.