UDC 004.415.53
Preliminary Communication

MATT: Multi Agents Testing Tool Based Nets within Nets

Sara Kerraoui *kerraouisara@hotmail.fr*
Faculty of Sciences- Department of Computer Sciences
University 20 August 1955- Skikda, Algeria.

Yacine Kissoum *kissoumyacine@yahoo.fr*
Faculty of Sciences- Department of Computer Sciences
University 20 August 1955- Skikda, Algeria

Mohammed Redjimi *medredjimi@gmail.com*
Faculty of Sciences- Department of Computer Sciences
University 20 August 1955- Skikda, Algeria

Moussa Saker *sak_moussa@yahoo.fr*
Faculty of Engineering- Department of Computer Sciences
University Badji Mokhtar- Annaba, Algeria

Abstract

Testing is a software development activity, devoted to evaluating product quality and improving it by identifying defects and problems. Concerning multi agent systems, testing is a challenging task, which asks for new testing techniques dealing with their specific nature. The techniques need to be effective and adequate to evaluate agent's autonomous behaviors and build confidence in them.
The "Model Based Testing" (MBT) is a technique particularly interested among all existing solutions of tests. This latter is based on a system model, which produces abstract test cases. To run these last ones against systems under test, the abstract test cases have to be transformed to concrete ones.
As part of this effort, we propose a model based testing approach for multi agent systems based on such a model called Reference net, where a tool, which aims to providing a uniform and automated approach is developed. The feasibility and the advantage of the proposed approach are shown through a short case study.
Keywords: Multi Agent Systems, Nets within nets, Reference nets, Renew, Model Based Testing.

1. Introduction

Agents and multi agent systems are currently developed in different active areas focusing mainly on architectures, protocols, frameworks, messaging infrastructures and community interactions. The obtained results show that these systems are undoubtedly more than a promising approach to complex software development.

However, testing is required to build confidence into the working of them, but testing of multi agent systems is a challenging task for several reasons, principally [1]:

- The increased complexity related with the multiple distributed processes running autonomously and concurrently making the system non-deterministic.
- The big quantity of data manipulated by a large number of agents (the systems can be made by thousands and more of agents, each using its own data).
- The effect of the irreproducibility: several executions of the same system with the same inputs will cannot lead with the same state making the looking for errors difficult.
- The agents are autonomous; they cooperate, interact and communicate with other agents by message passing or blackboard instead of invocation methods. Therefore, the multi agent system is non-deterministic and it is not possible to use directly the existing object-testing methods.

One of the new approaches to meet the challenges imposed on software testing is the Model Based Testing technique (MBT) [2], [4]. It has recently gained attention with the popularization of models in software design and development. In these techniques, a model of the desired behavior of the system under test is used for automated generation of test cases. Automated test cases generation is attractive because it has the potential to reduce the time required for testing. Test cases derived from the model are collectively known as abstract test cases and cannot be directly executed against a system under test. They have to be transformed to concrete test cases to communicate directly with the system under test. Our work comes in this context.
In fact, we propose a model based testing approach for multi agent systems based on models called reference nets [5], [6]. Reference nets are based on the nets within nets paradigm that generalizes token to data types and even nets. This model is renowned for its perfect adhesion to the mechanism of composition in multi agent systems.

The framework MULAN (MULti-Agent Network) [7], which is based on the visual programming concept of petri net gives tools for the modeling of agent application and bridges the gap between modeling and programming on the one side, and modeling and verification on the other side.
Our main contribution can be summarized as follow:

- To minimize the initial test effort by automatic generation of test cases supported mainly by the model used for modeling.
- To propose another way to concretize abstract test cases by instrumented model.

The remaining part of this paper is organized as follow: section 2 surveys the state of the art of Multi-Agent System (MAS) testing approaches. Section 3 deals with a short description of the paradigm of reference nets. Section 4 describes the proposed approach. Section 5 presents a case study. Finally, section 6 concludes this work with discussion about issues and future works.

2. Background and related works

Usually, the Multi agent systems can be considered at algorithmic, class, agent, society, and system levels of abstraction. These levels are defined as follow [2], [8].

- At the algorithmic level, we consider the code at routine level. This step deals only with the manipulation made within a routine and data. This is comparable to normal code testing with conventional imperative languages.
- At the class level, the interactions of routines and data that are encapsulated within a class are considered. This level leads with the object-oriented community. It has valorizing the emergence of the JUnit [9] testing framework.
- The interactions of groups of cooperating classes are considered at the agent level. At this step, it is possible to test the integration of the different modules inside an agent, to test agents' capabilities to achieve their goals and to sense and effect the context. There are several works in this scope; Tiryaki et al. [10] for example, proposed a test-driven MAS development approach supporting iterative and incremental MAS construction called SUnit, which was built on top of JUnit.
  Coelho et al. [11] proposed a framework for MAS unit testing similar to SUnit, but uses Mock Agents. Houhamdi [12] introduces a suite test derivation approach for agent testing that takes goal-oriented requirements analysis artifact as the core elements for test case derivation.
- The society level consists of the interactions of the overall results of different agents. The society level testing is a kind of integration testing and the integration strategy depends on the agent system architecture where agent dependencies are usually in terms of communications and sometimes environment mediated interactions could be present. Integration testing involves making sure that an agent works properly with the agents that have been integrated before it and with the agents that are in agent testing phase. In this context comes the Padgham et al. [13] work: they use design artifacts from the Prometheus design process (e.g., agent interaction protocols and plan specification) to provide automatic identification of the source of errors detected at the run-time process. Rodrigues et al. [14] propose to exploit social conventions, i.e. norms and rules that prescribe permissions, obligations and/or prohibitions of agents in open MAS to integration test.
- The system level contains all code from all classes and main program necessary to run the entire system. Explicitly, agents may operate correctly when they run alone but incorrectly when they are put together. We find in this level: De Wolf et al. [15] approach in which they propose an empirical analysis approach combining agent-based simulations and numerical algorithms for analyzing the global behavior of a self-organizing system. In [16], Houhamdi and Athamena used a suite test derivation approach for system testing.

Most of the existing research works on testing the MAS focuses primarily on the agents and integration level of agents. Our approach considers all the previous test levels (from the system to the algorithmic levels) as shown Figure 1.

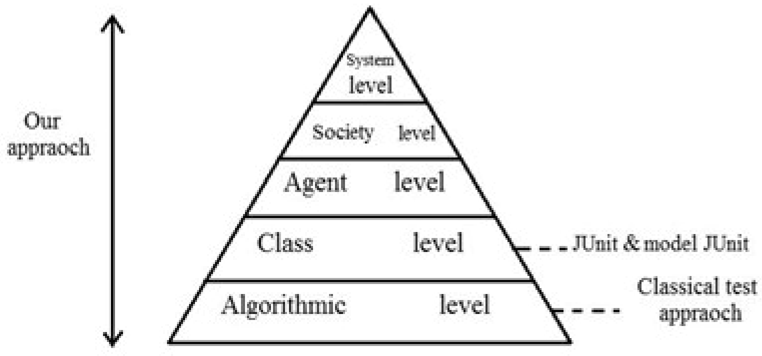


Figure 1. The scope of our approach.

3. Nets within nets

The paradigm of nets within nets [17] is an expressive high-level petri nets that allows nets to be nested within nets in dynamical structure. In other word: it formalizes the aspect that tokens of a Petri nets can be data types and even nets. A short introduction of the implementation of certain aspects of nets within nets called Reference nets [5], [6] will be given in subsequently.

3.1. Reference nets

Reference nets are a graphical notation especially well-suited for the description and execution of complex and concurrent processes. As for other net formalisms, there exist tools for the simulation of reference nets called Renew (for REference NEt Workshop) [6]. The reference nets extend black and colored Petri nets by means of net instances, nets as token objects, communication via synchronous channels and different arc types. Some definitions of these extensions are given in [18], [19].

3.2. Multi agent nets

The MULAN (MULti-Agent Network) architecture [7] is based on the nets within nets paradigm and is used to describe the natural hierarchies in a multi agent system. Mulan is implemented in Renew [6] and has the general structure as depicted in Figure 2. Each box describes one level of abstraction in terms of a system net.
The net in the upper left side of Figure 2 describes an agent system, which places contain agent platforms as tokens. The transitions describe communication or mobility channels, which build up the infrastructure. By zooming into the platform token on place p3, the structure of a platform becomes visible. The central place agents host all agents, which are currently on this platform. Each platform offers services to the agents. Agents can be created (transition *new*) or destroyed (transition *destroy*). Agents can communicate by message exchange. Two agents of the same platform can communicate by the transition *internal communication*. *External communication* only binds one agent, since the other agent is bound on a second platform somewhere else

in the agent system. In addition, mobility facilities are provided on a platform: agents can leave the platform via the transition *send agent* or enter the platform via the transition *receive agent*.

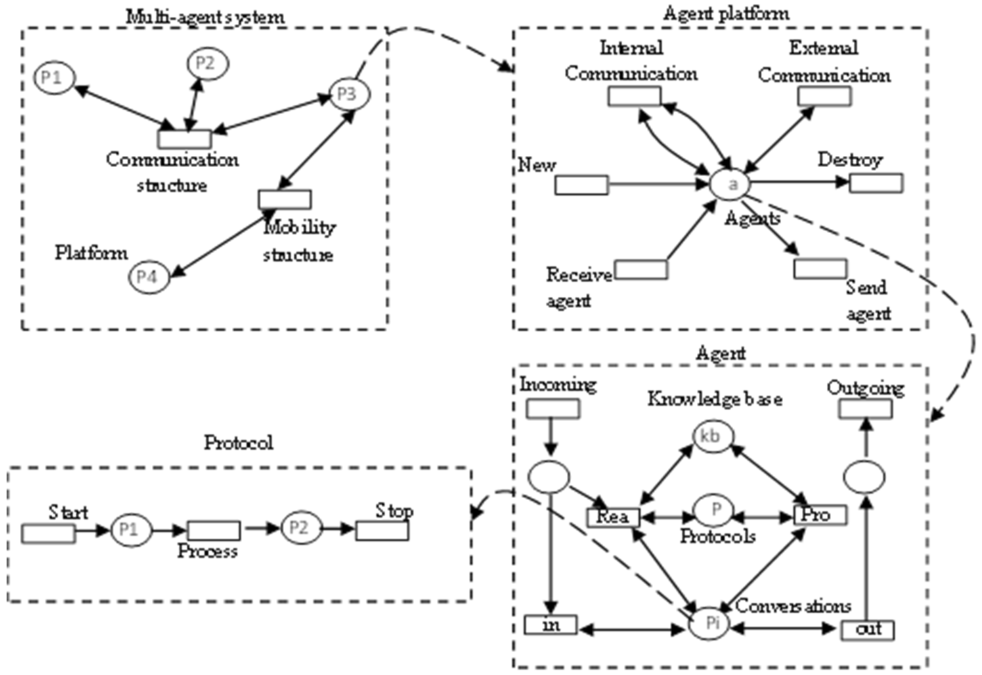


Figure 2. Agent systems as nets within nets (Source [18]).

Agents are also modeled in terms of nets. They are encapsulated, since the only way of interaction is by message passing. Agents can be intelligent, since they have access to a knowledge base. The behavior of the agent is described in terms of protocols, which are again nets. Protocols are located as templates on the place protocols. Protocol templates can be instantiated, which happens; for example if a message arrives. An instantiated protocol is part of a conversation and lies in the place conversations.

4. Proposed approach

Model-based testing (MBT) can be summarized in one sentence; it is essentially a technique for automatic creation of test cases from specified software model. It, usually, means functional testing for which the test specification is given as a test model. The test model is derived from the system requirements.
As mentioned in section 2, multi agent systems can be considered, from tester's point of view, as a number of different levels of abstraction: the algorithmic level, class, agent, society and system levels. Traditional test techniques (functional, structural and non-regression) have been fully implemented for the first level of abstraction. Furthermore, the need of a framework to support the development of automated tests

was perceived by the object-oriented community and it has led to the emergence of the Java Unit Testing (JUnit) frameworks, which is considered as reference of unit testing in Java [9]. Its effectiveness encouraged many researchers to propose extensions capable of supporting other application types: DBUnit (for databases) and NModel (for applications written in C #) for model Mark Utting proposed based testing technique, an extension of JUnit, called Model JUnit [20]. Its principle is to write simple finite state machine (FSM) models or extended finite state machine (EFSM) models as Java classes, then generates tests from those models.

The approach discussed in this paper was inspired from Model-JUnit, but it differentiates itself by the fact that it is dedicated to multi-agent applications. Moreover, while Model-JUnit utilizes finite state machine as model, our approach is based on reference nets, which have a graphical representation that provides a flexible modeling approach where tokens can be Java-objects and nets can be regarded as objects. Finally, in Model JUnit, the tester should write test cases to be executed by using JUnit. In our approach, the test cases are generated automatically from the model and is enforceable dynamically and simultaneously with the execution of the system under test.

The Figure 3 summarizes the general architecture of our approach. It contains a series of four stages.

As shown by (1) the modeling phase is the first step in model based testing technique. In this phase, we represent the abstract specification in the form of a model. The model is built to represent the intended behavior of the system under test. However, to verify the validity of the model, phase (2) is mandatory. The tester (the modeler) has to simulate its model for each possible scenario, in certain cases corrects, and refines progressively the abstract test model until he is satisfied that the model meets the initial specifications. The no detected errors in the validation phase can be detected during the phase of execution of test cases. Phase (3) represents the test cases generation and their concretization.

It is the most delicate step and constitutes our main contribution. It cannot be conducted without the source code of the system under test and is directed under the responsibility of the tester. It takes as input an information file extracted from the system under test and another from the test model: All the methods present in the source code and all the transitions in model extracted as XML format or more accurately (PNML RefNet format) file and, of course, the scenario that the tester wants to check. Such scenario is built according to the specification of the test and the levels of the desired test (agent, society or system). While adopting a test case design technique called "error-guessing ", which a technique is based on the ability of the tester to draw on his past experience, knowledge and intuition to predict where bugs will be found in the system under test. The output of instrumentation is then a new version of the model. The last step is the execution phase (4). During this stage, the tester can invoke the execution of the instrumented version of the model. The results obtained following the execution of a sequence of test will be compared to the expected results and a verdict is constituted.

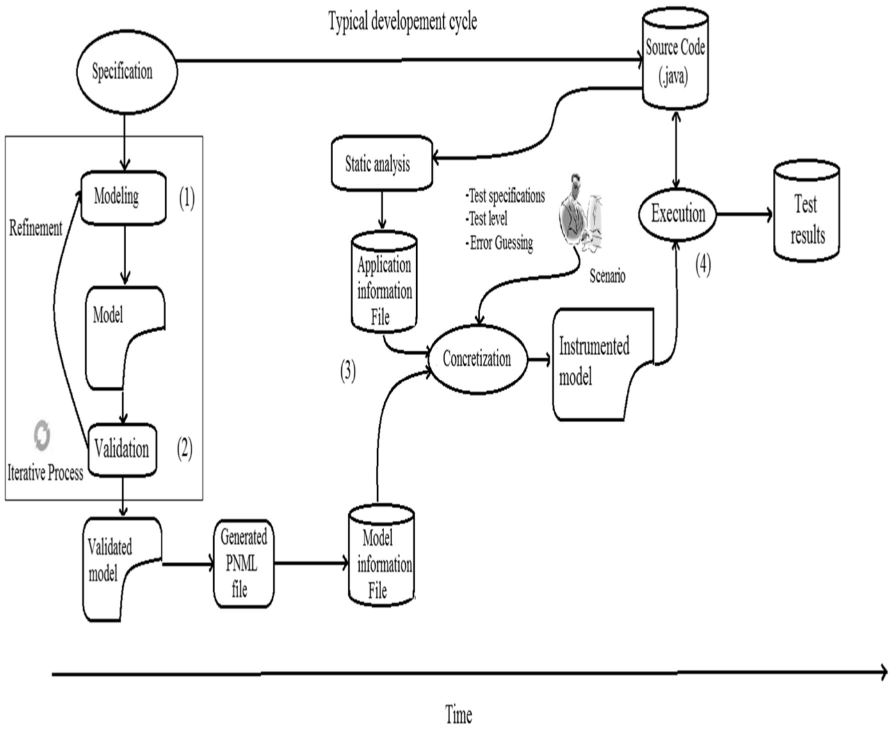


Figure 3. Proposed approach.

To better understand the proposed approach and to explain the different stages, we introduce as case study the well-known producer-consumer model.

5. Case study

The producer-consumer model has been used so abusive in computer literature, for example, to solve synchronization problems, coordination and communication. The Figure 4 describes our case study at an abstract level.
The example is constituted from a producer, which transmits data to a consumer. It is imperative to point out that this example works under the following assumptions:

- Consumers must collect original data disassembled by the producer before transmission,
- The principle of transmission is according to the stop-and-wait protocol (Asynchronous mode). That is, the producer transmits a data packet at a time and must wait for an acknowledgment.
- The principle of transmission is according to the stop-and-wait protocol (Asynchronous mode). That is, the producer transmits a data packet at a time and must wait for an acknowledgment.

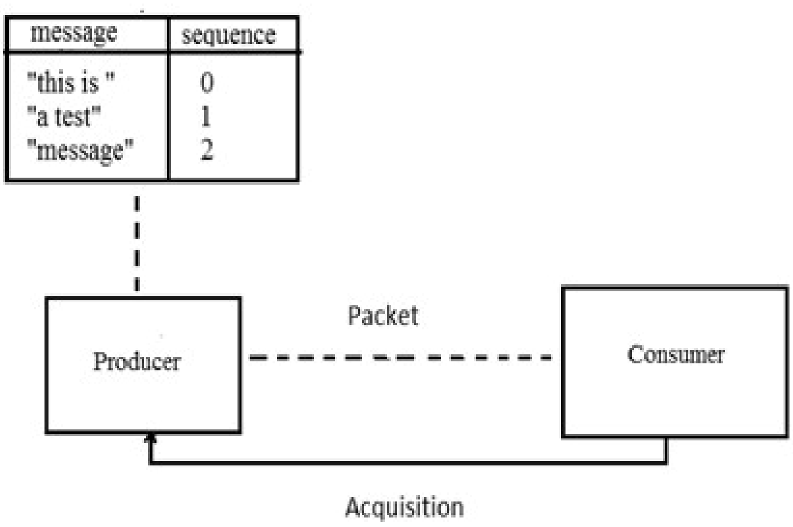


Figure 4. Producer-consumer case study.

The steps sequences of the proposed approach are as follow:

5.1. Building the abstract test model

In the MBT process, we represent the abstract specifications in the form of a model. This may lead to discover inconsistencies in the specification, thus, leading to correct code from the beginning. Models of the system are useful for predicting the outputs of the system, which allows test case to be generated. One of major drawbacks of the model based testing techniques is the cost of building the abstract test model of the system under test. In fact, the behavior of the system can be described using a variety of different model types and a few of which make good models for testing, but the choice depends on aspects of the system under test. To the best of our knowledge, existing model based testing techniques for multi agent systems do not cover every aspect of multi-agent systems such as dependencies and interactions. The proposed approach covers such interactions as well, because it is based on the Reference nets model. The latter is based on the nets within nets paradigm, which is renowned for:

- First, for its perfect adhesion to the mechanism of composition in multi agent systems: agent protocols are composed to agent behavior, agent behavior is composed to agents, agents are composed to groups, groups are composed within agent platforms and platforms are composed to agent systems.
- Furthermore, for its unifying framework MULAN based on the visual programming concept of petri net for the modeling of agent application in an elegant and intuitive manner without losing formal accuracy.
- Finally, a model built upon a formalism that has a formal semantics to support verification, simulation and execution.

In Figure 5, the snapshot (a) represents the main net (system level); it creates the producer and the consumer nets. Snapshot (b) shows the structure of our autonomous agents. Finally, the snapshot (c) describes, respectively, the producer and the consumer protocols. After the producer protocol starts the transition: *transmit()* produces a performative containing a message "this is the test message" which represents the product sending and that is directed to the consumer. The performative will be sent over the p*: send()* transition; subsequently the protocol is blocked waiting an answer message. The blocking behavior is necessary to simulate a synchronous communication between producer and consumer. An arriving acknowledges confirmation to p: *ack(j)* enables the transition acknowledges received. After occurrence of that transition, the protocol is not blocked any further and the producer agent is now able to reproduce another item. The protocol net that models the consume behavior of the consumer agent is selected by the agent's main page to process an incoming performative item from the producer agent. It can now acknowledge the reception, consume the item and wait for another product.

5.2. Validating the model

Because this abstract formal test model is derived manually out of the systems requirements specification, the validation of the test model against system requirements specification should be done first in order to find major errors in the test model. With MBT approach, if some errors remain in the model they are very likely to be detected when the generated test is run against the system under test. Renew relies entirely on simulation to explore the properties of a net, where the tester can dynamically and interactively explore the state of the simulation. In fact, at this stage, the tester can imagine different execution scenarios by acting, for example, on the number of agents or the number of objects (resources).
In other words, while simulating the system net, the tester has the opportunity to correct and to refine progressively the abstract test model. Such robust and easy to use tool reduces considerably the large initial effort in term of person-hours required mainly in constructing and validating the testing model.

5.3. Test cases generation and concretization

Test cases are generated automatically from the behavioral model, which is given as input to the test generators. In the literature [21], the test cases generation approaches have suffered from the combinatorial explosion problem of the number of test cases. For economic reasons, it should be reduced to a minimum. For quality reasons, it must be sufficiently high as to reduce the number of remaining failures in the field to an acceptable number. To cope with this explosion, we have opted for the well-known test cases generation techniques. In our case, let us consider, for instance, the class partitioning and limit testing techniques.

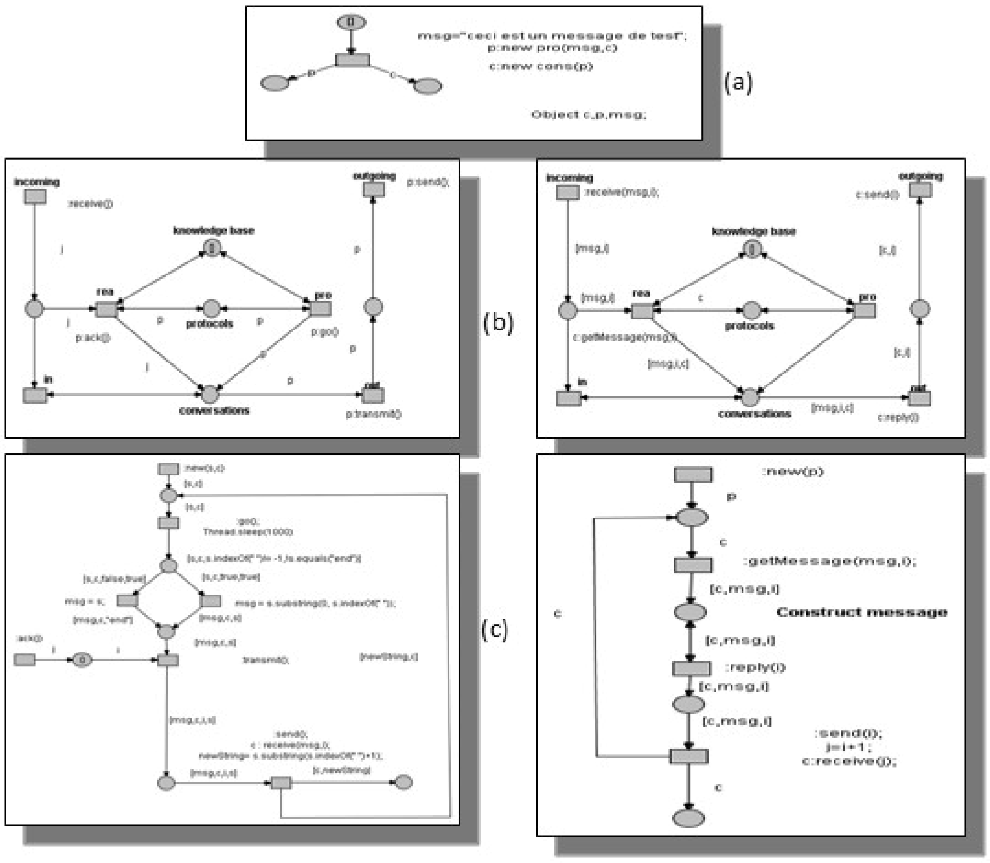


Figure 5. The modelling phase.

The strategy of such technique can be summarized as follow:

- A- Identification of operational variables,
- B- definitions of domains (and equivalence classes) of variables,
- C- Development of an operational relationship between the variables; modeling different system responses in a variant of the decision table,
- D- Development of test cases for the variants.

A- Identification of operational variables:
Msg: variable modeling the packet to be transmitted,
P: object modeling the producer,
C: object modeling the consumer.

B-definition of domains:
Msg :{ Ok, Empty, Not string value},
P :{ Ok, Null},
C: {Ok, Null}.

C-Decision table:
Number of test cases: 3*2*2= 12(see Table 1).

Test case	Msg	P	C	Result
1	Ok	Ok	Ok	Successful transmission
2	Ok	Null	Ok	Error
3	Ok	Ok	Null	Blocked
4	Empty	Ok	Ok	Blocked
5	Empty	Null	Ok	Blocked
6	Empty	Ok	Null	Blocked
7	Not string value	Ok	Ok	Error
8	Not string value	Null	Ok	Error
9	Not string value	Ok	Null	Error
10	Ok	Null	Null	Blocked
11	Empty	Null	Null	Blocked
12	Not string value	Null	Null	Error

Table 1. The decision table.

D-Development of the test cases:
-A test case is “true” when it meets all requirements of the variants on the values of the variables.
-A test case is “false” when it violates the requirements of the variants.

To increase the effectiveness of the proposed approach and to compensate the inherent incompleteness of equivalence partitioning and boundary value analysis, an error guessing test case design technique is used. Such a technique is based on the ability of the tester to draw on his past experience, knowledge and intuition to predict where bugs will be found in the system under test. By doing so, the number of possible test traces (scenarios) is reduced. Here depending on abstract test level the tester wants to check, additional information can be extracted from design artifacts: state charts for algorithmic level, class diagram for class level, sequence diagram for agent level, collaboration diagram for society level and uses case diagram for system level.

Concerning concretization, the generated test cases are abstract like the test model. Thus, significant information is missing in the generated test cases to be executable with the concrete system under test. Therefore, these test cases have to be concretized. In other words, this step acts as a translator which bridges the abstraction gap between the test model and the system under test by adding missing information and translating entities of the abstract test case to concrete constructs of the test platform’s input language.
This last one is a difficult task because it is done generally by applying different algorithms. Furthermore, the model based testing solution ensures code coverage as shown Figure 6.

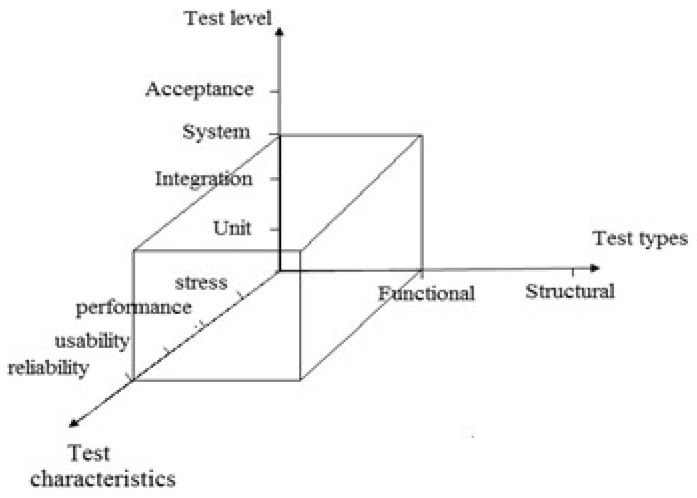


Figure 6. The scope of model based testing approach [22]

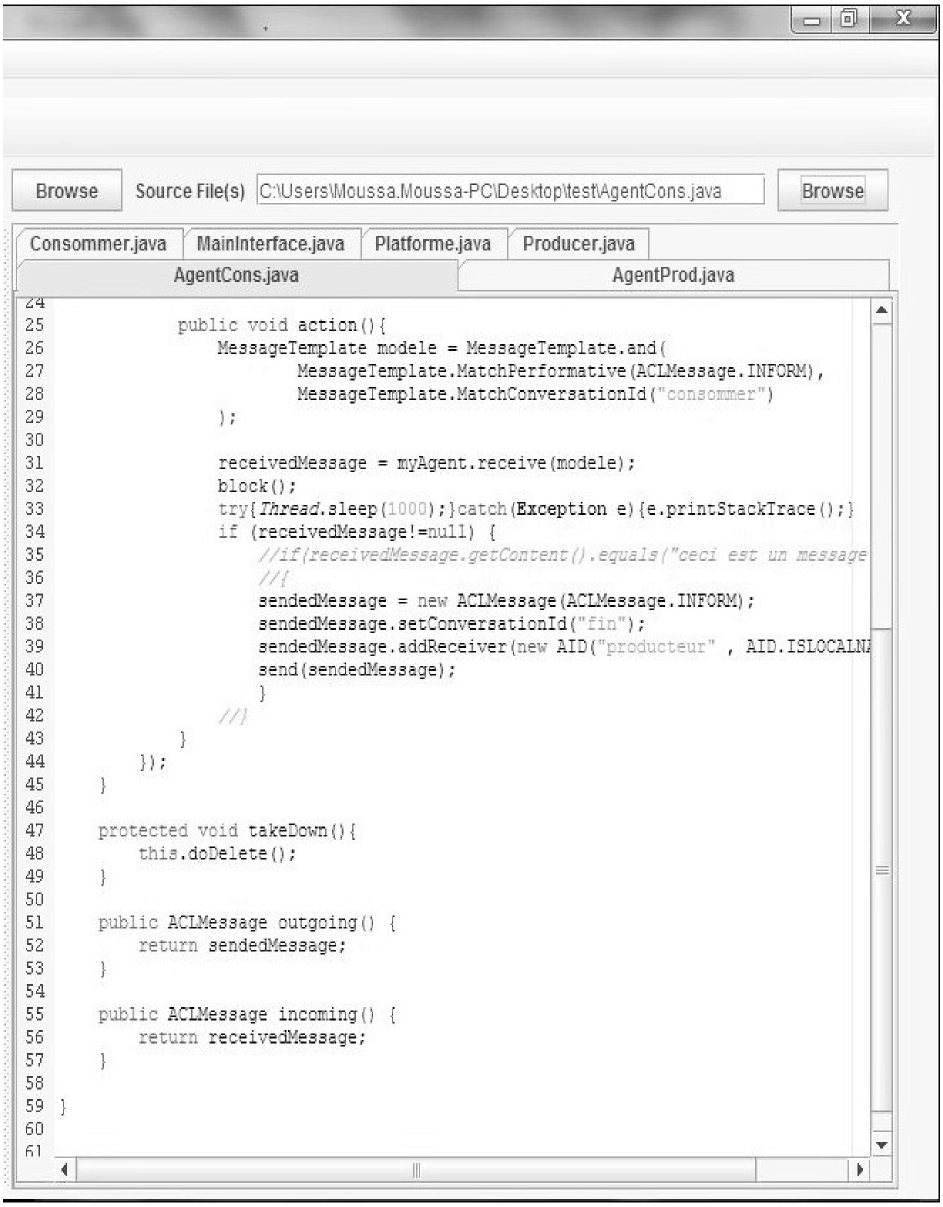
However, we make changes in the code of system under test by Adding missing information (instrumentation of code). We come across the following problems: over load, stress, performance, volume, security, usability, storage.

To remedy these problems, we have proposed to do the opposite (why not instrumentation of the model?), since, reference nets are themselves Java objects: Making calls from Java code to net is easy as to make calls from nets to Java code. This phase begins with the launch of the test monitor, which we have developed.

So, before realizing this phase, a preparative work must be done. This is the static analysis.

Concerning system under test (source code), information is extracted from the agent's internal structures by parsing them. This information includes details as the identification of agents, their plans, the names of classes (the agent is composed from the agent base class and several behavior classes) together with the names of each of member methods, their parameters, and their types. Those information are then stored in a so-called application information file as shown in Figure 7.

Concerning the model, all nets are exported into XML format or more accurately PNML RefNet format (see Figure 8).Once all inputs (system information file, model information file and the scenario of test) are available, the test monitor proceeds to the instrumentation of scenario of test by adding a certain set of specific routines in model. More precisely, the instrumentation is done by selecting the methods we want to test from application information file and insert each method in the most suitable transition in model information file as is shown in Figure 9.The output is a new version of the model.

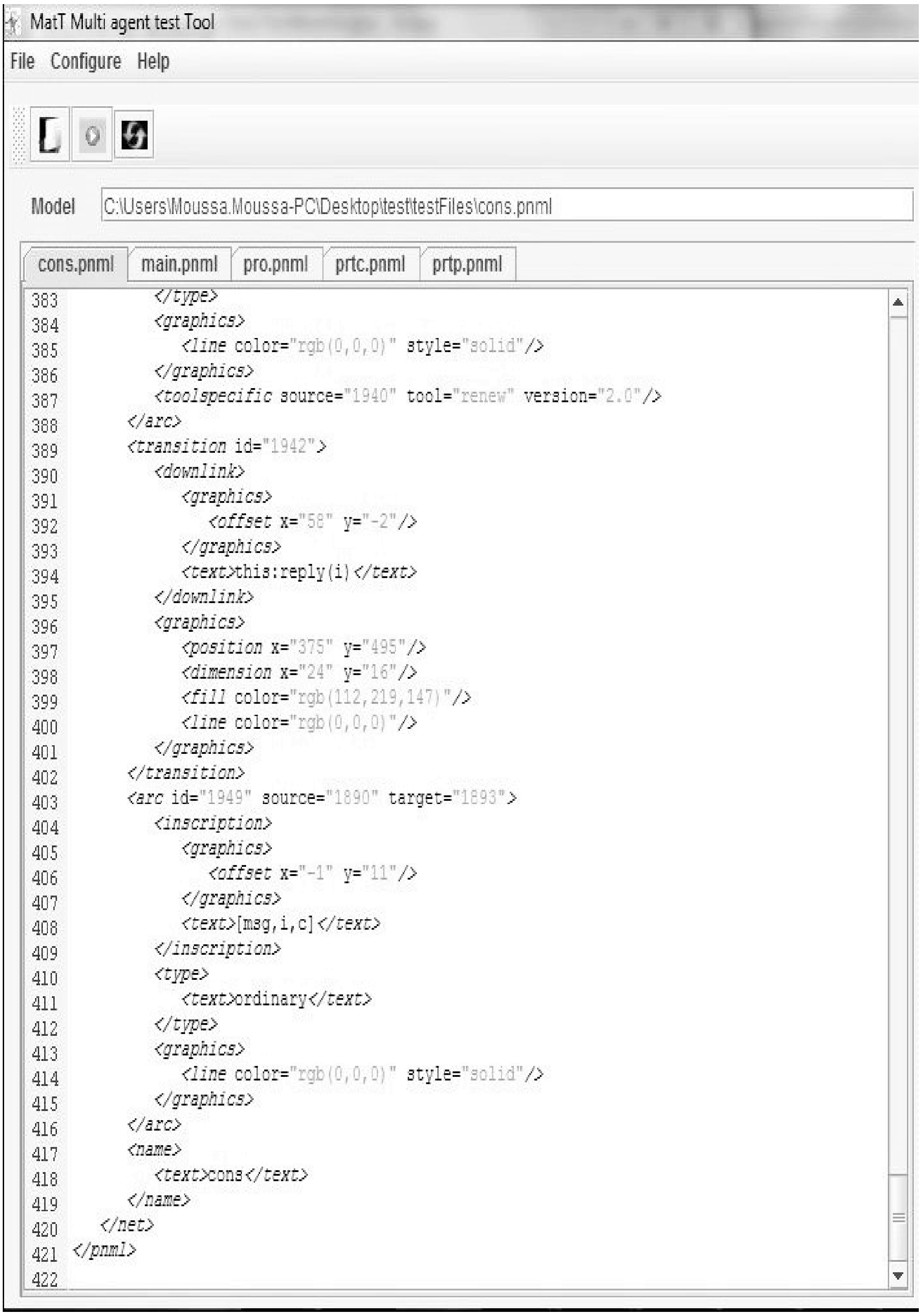


Source File(s) C:\Users\Moussa.Moussa-PC\Desktop\test\AgentCons.java

```
        public void action(){
            MessageTemplate modele = MessageTemplate.and(
                    MessageTemplate.MatchPerformative(ACLMessage.INFORM),
                    MessageTemplate.MatchConversationId("consommer")
            );

            receivedMessage = myAgent.receive(modele);
            block();
            try{Thread.sleep(1000);}catch(Exception e){e.printStackTrace();}
            if (receivedMessage!=null) {
                //if(receivedMessage.getContent().equals("ceci est un message
                //{
                sendedMessage = new ACLMessage(ACLMessage.INFORM);
                sendedMessage.setConversationId("fin");
                sendedMessage.addReceiver(new AID("producteur" , AID.ISLOCALN
                send(sendedMessage);
                }
            //}
        }
    });
}

protected void takeDown(){
    this.doDelete();
}

public ACLMessage outgoing() {
    return sendedMessage;
}

public ACLMessage incoming() {
    return receivedMessage;
}

}
```

Figure 7. Result of static analysis of source code.

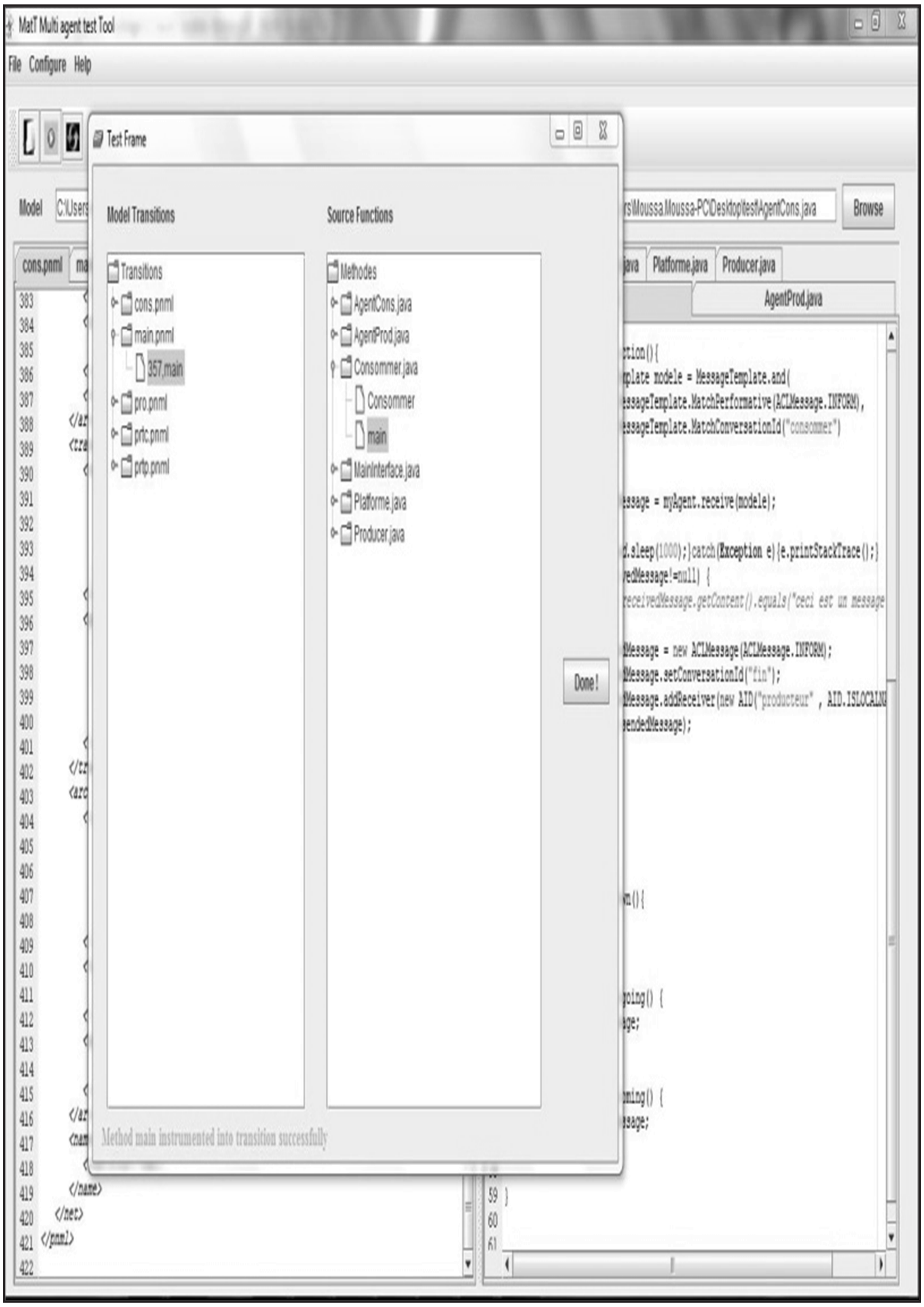
In particular, the instrumentation will allow the tester to follow the execution of the traces of the various operations as the progresses of the simulation of the model.
By doing this, the tester can, at this stage, accept, reject or modify the generated variable values. Figure10 shows the instrumented version of model where all adding instruments are encircled. The reader can easily compare the original figures of the model to the figure 10. Finally, because this operation (instrumentation) is derived automatically, this significantly reduces the testing effort.



C:\Users\Moussa.Moussa-PC\Desktop\test\testFiles\cons.pnml

```
          </type>
          <graphics>
             <line color="rgb(0,0,0)" style="solid"/>
          </graphics>
          <toolspecific source="1940" tool="renew" version="2.0"/>
       </arc>
       <transition id="1942">
          <downlink>
             <graphics>
                <offset x="58" y="-2"/>
             </graphics>
             <text>this:reply(i)</text>
          </downlink>
          <graphics>
             <position x="375" y="495"/>
             <dimension x="24" y="16"/>
             <fill color="rgb(112,219,147)"/>
             <line color="rgb(0,0,0)"/>
          </graphics>
       </transition>
       <arc id="1949" source="1890" target="1893">
          <inscription>
             <graphics>
                <offset x="-1" y="11"/>
             </graphics>
             <text>[msg,i,c]</text>
          </inscription>
          <type>
             <text>ordinary</text>
          </type>
          <graphics>
             <line color="rgb(0,0,0)" style="solid"/>
          </graphics>
       </arc>
       <name>
          <text>cons</text>
       </name>
    </net>
 </pnml>
```

Figure 8. Result of static analysis of model.

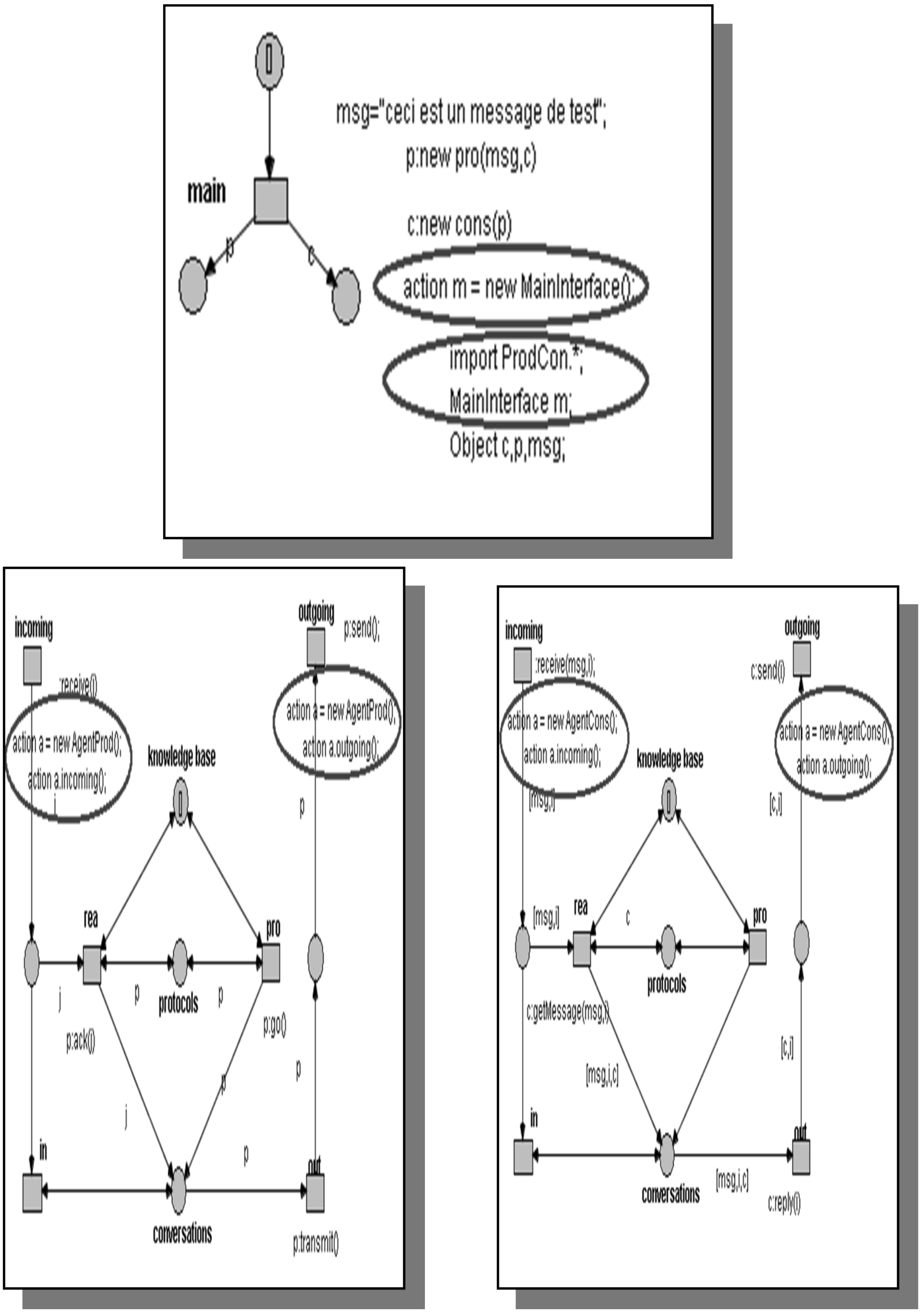


Method main instrumented into transition successfully

```
        </name>
    </net>
</pnml>
```

```
ction(){
plate modele = MessageTemplate.and(
essageTemplate.MatchPerformative(ACLMessage.INFORM),
essageTemplate.MatchConversationId("consommer")


essage = myAgent.receive(modele);

d.sleep(1000);}catch(Exception e){e.printStackTrace();}
vedMessage!=null) {
receivedMessage.getContent().equals("ceci est un message

dMessage = new ACLMessage(ACLMessage.INFORM);
dMessage.setConversationId("fin");
dMessage.addReceiver(new AID("producteur" , AID.ISLOCALNA
sendedMessage);

n(){

going() {
ge;

ming() {
ssage;

}
```

Figure 9. Instrumentation of model.



```
msg="ceci est un message de test";
p:new pro(msg,c)
c:new cons(p)
action m = new MainInterface();
import ProdCon.*;
MainInterface m;
Object c,p,msg;
```

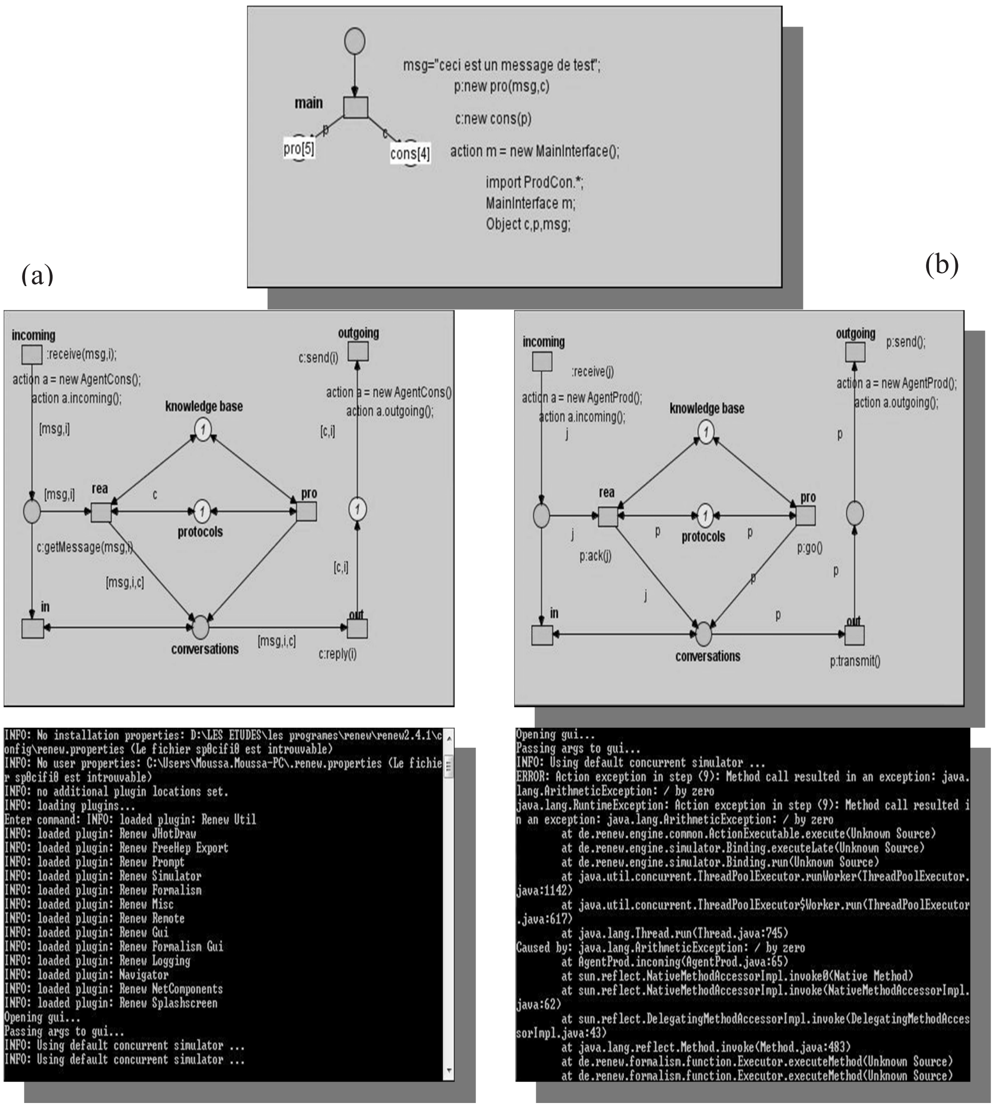
```
:receive()
action a = new AgentProd();
action a.incoming();
p:send();
action a = new AgentProd();
action a.outgoing();
p:ack(j)
p:go()
p:transmit()
```

```
:receive(msg,j);
action a = new AgentCons();
action a.incoming();
c:send()
action a = new AgentCons();
action a.outgoing();
c:getMessage(msg,j)
c:reply()
```

Figure 10. Instrumented version of the model

5.4. Execution and evaluation

Now, instrumented version of the model under test is ready for execution with the desired test case. The results obtained following the execution of a sequence of test will be compared to the expected results and a verdict is constituted. Figure11 presents the execution of producer consumer example.
The execution of instrumented model calls concerned function by test in application. If the test passes correctly, the execution of model continues to a new place or transition as shown in snapshot (a). Otherwise, an error message is displayed on the command line in snapshot (b).



```
INFO: No installation properties: D:\LES ETUDES\les programes\renew\renew2.4.1\c
onfig\renew.properties (Le fichier sp0cifi0 est introuvable)
INFO: No user properties: C:\Users\Moussa.Moussa-PC\.renew.properties (Le fichie
r sp0cifi0 est introuvable)
INFO: no additional plugin locations set.
INFO: loading plugins...
Enter command: INFO: loaded plugin: Renew Util
INFO: loaded plugin: Renew JHotDraw
INFO: loaded plugin: Renew FreeHep Export
INFO: loaded plugin: Renew Prompt
INFO: loaded plugin: Renew Simulator
INFO: loaded plugin: Renew Formalism
INFO: loaded plugin: Renew Misc
INFO: loaded plugin: Renew Remote
INFO: loaded plugin: Renew Gui
INFO: loaded plugin: Renew Formalism Gui
INFO: loaded plugin: Renew Logging
INFO: loaded plugin: Navigator
INFO: loaded plugin: Renew NetComponents
INFO: loaded plugin: Renew Splashscreen
Opening gui...
Passing args to gui...
INFO: Using default concurrent simulator ...
INFO: Using default concurrent simulator ...
```

```
Opening gui...
Passing args to gui...
INFO: Using default concurrent simulator ...
ERROR: Action exception in step (9): Method call resulted in an exception: java.
lang.ArithmeticException: / by zero
java.lang.RuntimeException: Action exception in step (9): Method call resulted i
n an exception: java.lang.ArithmeticException: / by zero
        at de.renew.engine.common.ActionExecutable.execute(Unknown Source)
        at de.renew.engine.simulator.Binding.executeLate(Unknown Source)
        at de.renew.engine.simulator.Binding.run(Unknown Source)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.
java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor
.java:617)
        at java.lang.Thread.run(Thread.java:745)
Caused by: java.lang.ArithmeticException: / by zero
        at AgentProd.incoming(AgentProd.java:65)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:483)
        at de.renew.formalism.function.Executor.executeMethod(Unknown Source)
        at de.renew.formalism.function.Executor.executeMethod(Unknown Source)
```

Figure 11. Execution and evaluation of results.

6. Conclusion and future works

In this paper, we have presented a model based testing approach for multi agent system using the paradigm of reference nets and we have used the producer consumer example as a case study.
An important argument for using the paradigm of nets within nets is that the modeling process concludes not only with a running system model, but also because it supports the whole steps of the model testing technique. This reduces greatly the major negative aspects of model based testing techniques and support testers in creating and executing tests in a uniform and automatic way.
This paper reflects our experiences with reference nets and hence renew tool. We think that it is a robust and easy to use tool, which reduces considerably the large initial effort in term of person-hours required mainly in constructing and validating of the testing model. In fact, one of major drawbacks of the model based testing techniques is the cost for the building of the abstract test model of the system under test.
This paper represents a different approach compared to other works; first the scope of our approach touches all test levels (from the system to algorithmic levels).besides we solve the problem of combinatorial explosion in the number of test cases by adopting error guessing technique. Moreover, we have automated the concretization stage by instrumented model, finally and to validate our approach we have developed a test monitor that supports all phases of the testing technique based models: modeling, validation, implementation and execution. The test results generated by the test monitor facilitate the construction of a verdict about the application. Therefore, our approach can be a valid support for these kinds of test.
Nevertheless, the proposed testing approach has focused on a simple form of agent cooperation. It is our intention to build models for a number of sophisticated multi agent coordination.
Since our approach is based on formal models, automation of building and validating models will be another perspective.
A final perspective could involve the integration of structural coverage at our test monitor to test other characteristics of multi-agents applications (robustness, safety, and reliability).

References

[1] Z. Houhamdi, “Multi-Agent System Testing: A Survey.” *Int. Journal of Advanced Computer Science and Applications*, Vol. 2, No. 6, 2011.

[2] Y. Kissoum and Z. Sahnoun, “Formal specification and testing of multi agent systems.” in *8th Colloq. Africain sur la Recherche en Informatique (CARI’06)*, 2006.

[3] M. Broy et al., “Model-Based Testing of Reactive Systems: Advanced Lectures.” *LNCS*, Springer-Verlag New York, Inc. 2005.

[4] M. Utting, B. Legeard, “Practical Model-Based Testing: A Tools Approach.” *Morgan-Kaufmann*, ISBN 978-0-12-372501-1, 2007.

[5] O. Kummer, “Simulating synchronous channels and net instances.” *Workshop Algorithmen und Werkzeuge für Petrinetze*, Fachbereich Informatik, Universität Dortmund, 1998, pp 73–78.

[6] O. Kummer and F.Wienberg, “Reference net workshop (Renew).” Available:www.renew.de

[7] M. Duvigneau et al., “Concurrent architecture for a multi-agent platform.” in *Proc.Workshop on Agent Oriented Software Engineering (AOSE’02), LNCS 2585*, Springer Verlag, 2003.

[8] Y. Kissoum and Z. Sahnoun, “A formal approach for functional and structural test cases generation in multi agent systems.” in *5th IEEE/ACS Int. Conf. on Computer Systems and Applications (AICCSA’07),* 2007.

[9] E. Gamma, and K. Beck. (2000) “JUnit: A Regression Testing Framework”. available: http://www.junit.org

[10] A. M. Tiryaki et al., “SUnit: A unit testing framework for test driven development of multi-agent systems.” *AOSE'06 Proc.7th Int. Workshop on Agent-Oriented Software Engineering VII*, Springer, Berlin, 2007,.

[11] R. Coelho et al., “Unit testing in multi-agent systems using mock agents and aspects.” *Proc.Int. workshop on Software engineering for large-scale multi-agent systems,* ACM Press, New York, 2006 , pp. 83–90.

[12] Z. Houhamdi, “Test Suite Generation Process for Agent Testing.” *Indian Journal of Computer Science and Engineering,* Vol. 2, N °2, 2011.

[13] L. Padgham et al., “Adding debugging support to the Prometheus methodology.” *Engineering Applications of Artificial Intelligence,* Vol. 18, 2, 2005, pp. 173-190.

[14] L. Rodrigues et al., “Towards an integration test architecture for open MAS.” *1st Workshop on Software Engineering for Agent-Oriented Systems/SBES*, 2005, pp. 60-66.

[15] T. De Wolf et al., “Engineering self-organising emergent systems with simulation-based scientific analysis.” *Proc.4th Int. Workshop on Engineering Self-Organising Applications,* 2005, pp. 146-160.

[16] Z. Houhamdi, and B. Athamena, “Structured System Test Suite Generation Process for Multi-Agent System.” *Int. Journal on Computer Science and Engineering,* Vol.3, 4, 2011, pp.1681-1688.

[17] R. Valk, “Petri nets as token objects: An introduction to elementary object nets.” *Application and Theory of Petri Nets*, volume 1420 of LNCS*,* 1998, pp 1–25.

[18] M. Köhler et al., “Modeling the structure and behavior of petri net agents.” *J.M. ICATPN* 2001, LNCS 2075, 2001, pp 224–241.

[19] M. Köhler et al., “Modelling mobility and mobile agents using nets within nets.” *ICATPN* 2003, LNCS 2679, 2003, pp 121–139.

[20] Http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/, last accessed on March 15, 2009.

[21] G.J. Myers, “The art of software testing.” *John Wiley & Sons, Inc., Hoboken,* New Jersey, 2nd Ed. 2004

[22] S. Weibleder, “Test models and coverage criteria for automatic model based test generation with UML state machine.” Dissertation thesis, Mathematisch-Naturwissenschaftlichen Fakultät II, Humboldt-Universität zu Berlin, 2010, p 31.