People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University 20 Août 1955- Skikda
Faculty of Sciences
Computer Science department

Thesis

In order to obtain the diploma of

Master

Option: Advanced Software and Application Engineering

Model-Based Testing of Multi-Agent Systems

Presented by:

MEZDOUR Khaled

BOULBIR Karima

Framed by:

M. KISSOUM Yacine

Academic year 2021/2022

Dedications

We would like to dedicate our Master-thesis:

- To our family especially our parents whose unbelievable endurance, unconditional love, and untouchable devotion have been monumental;

-To all our brothers and sisters;

-To those who will be happy with this new goal in our study career;

-To all our best friends;

- To anyone who has ever taught us anything.

There are many friends and other family members who need to be listed for their part in this Master-thesis.

Finally, this Master-thesis is dedicated to all those who believe in the richness of learning, and, we would like also to dedicate this modest review to all those who have devoted their lives to bringing the faded light of ambiguity to the complete shininess of clarity.

Acknowledgements

In the name of Allah, The Most Beneficent and the Most Merciful.

All praises to Allah the Almighty for giving us the strengths, guidance, and patience in completing this Master-thesis. With His blessing, this Master-thesis is finally accomplished. First of all, there are a lot of people that helped us significantly throughout these years to reach the end of this beautiful journey. All those people were very influential and supportive, and we would like to thank them and show our appreciation for what they did. We would like to take this opportunity, first and foremost, to express our heartiest thanks and deep gratitude to our supervisor, **Mr. KISSOUM Yacine** *for his helpful guidance, valuable discussions, and support throughout this bibliographic research. We wish to express our gratitude to* **Mr. BENOUDINA Lazhar** *the Head of the Informatics Department, Faculty of science, Skikda University, for all kinds of official help, and for offering facilities and support to carry out this bibliographic research. We would like also to place on record our great appreciation to all our teachers at Skikda University, where we have studied an Advanced Software and Application Engineering specialty. These acknowledgments would not be complete without thought to our family. We want to especially express our deep gratitude to our dearest parents for their endless support. They have always been there for us during all these years of study and encouraged us to finish our Master's studies. Last but not the least; we would like to thank each and every member of our family, who spurred our efforts with their love and affection, inspiration, and care. To this end, we fully take all responsibility for any mistakes that may have occurred in this work.*

Abstract

This thesis uses a method that encompasses all levels of higher agent abstraction to address the challenge of formal model-based testing for multi-agent systems. The agent level, the agent company level, and the system level are the three levels of abstraction. The proposed method is ethical when it comes to model-based testing. By establishing a process that incorporates requirement management, test case creation automation, and test case execution. We first propose an agent-based system that uses reference networks to represent a mobile agent. The example was created using the MULAN (multi agent system net) architecture and the Java programming language. Second, we provide a model-based (network-to-network) testing approach that works in conjunction with the JUNIT tool for developing and testing multi-agent systems. Unlike other test methodologies, where instrumentation is primarily related to the program's source code, our approach keeps the tested application intact by instrumenting at the model level, allowing it to retain its initial behavior.

Keywords: Agents, Multi-Agent Systems, Model-Based Testing, Reference nets, nets within nets

ملخص

تستخدم هذه الأطروحة طريقة تشمل جميع مستويات تجريد العامل الأعلى لمواجهة التحدي المتمثل في الاختبار الرسمي القائم على النموذج للأنظمة متعددة العوامل. مستوى الوكيل ومستوى شركة الوكيل ومستوى النظام هي مستويات التجريد الثلاثة. الطريقة المقترحة أخلاقية عندما يتعلق الأمر بالاختبار القائم على النموذج. من خلال إنشاء عملية تتضمن إدارة المتطلبات وأتمتة إنشاء حالة الاختبار وتنفيذ حالة الاختبار. نقترح أولاً نظامًا قائمًا على الوكيل يستخدم شبكات مرجعية لتمثيل وكيل المحمول. تم إنشاء المثال ثانيًا، نقدم نهج اختبار قائم على النموذج (من شبكة إلى .Java (شبكة نظام متعددة الوكلاء) ولغة برمجة MULAN باستخدام بنية لتطوير واختبار الأنظمة متعددة الوكلاء. على عكس منهجيات الاختبار الأخرى، حيث JUNIT شبكة) يعمل جنبًا إلى جنب مع أداة ترتبط الأجهزة بشكل أساسي بالشفرة المصدر للبرنامج، فإن نهجنا يحافظ على التطبيق الذي تم اختباره سليماً من خلال الأدوات على .مستوى النموذج، مما يسمح له بالاحتفاظ بسلوكه الأولي

الكلمات المفتاحية

الوكلاء، الأنظمة متعددة العوامل، الشبكات المرجعية للاختبار القائم على النموذج، الشبكات داخل الشبكات

Table of contents

Liste of figures

Liste of tables

General introduction

With understanding of the challenges experienced in the creation of significant IT projects, the "software crisis" erupted around the end of the 1960s. Since Alain Turing devised a "test" dubbed "imitation game" in 1950 to evaluate if machines could think, a new profession, software engineering, has grown out of this finding. Software testing has progressed significantly, especially since the introduction of multi-agent systems. The MULAN technique to model-based testing of multi-agent systems is the focus of this thesis.

Modeling is the initial stage in the model-based testing process, and it is the foundation of our work. Because of the triggering nature of the entire test process, this step is the most challenging. Because of the triggering nature of the entire test procedure, this is the most challenging step. We propose a system based on agents. The entire system is represented using the MULAN architecture (Multi agent Nets), which is one of the nets-within-nets paradigms. The Renew tool validates the model that has been created.

After that, we're ready for the test, which is why our second contribution consists of a test strategy suggestion that attempts to:

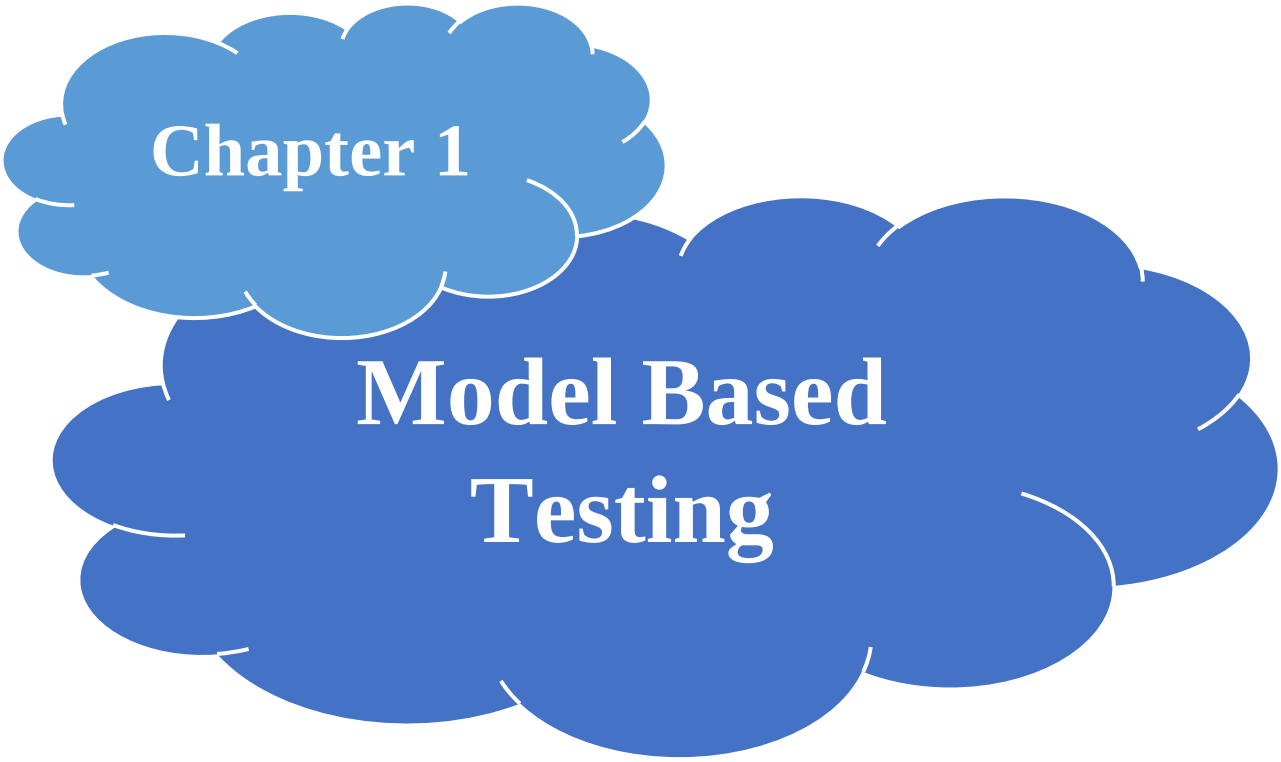
Automate the step of concretization of abstract test cases (derived from test models) into concrete test cases, reducing the initial testing labor (very close to the programming language used for the development of the system under test).

Our work is divided into four parts. The first chapter introduces the test in the software development life cycle, as well as the definitions, different types of software tests, and the test base model.

The second chapter is dedicated to the depiction of multi-agent systems and, as a result, the MAS's many tests' levels.

The Petri nets was described in the third chapter in terms of its formal and informal definitions, as well as its quality, features, and certain extensions.

We created a model and implemented our case study in the fourth chapter.


Chapter 1
Model Based Testing

1. MODEL BASED TESTING

1.1. Introduction

In our daily lives, software has become necessary and ubiquitous, and it is frequently employed to do vital activities.

Unfortunately, these systems have frequently failed, often resulting in substantial damage and human life loss; well-known instances include:

- (October 2007, South African army): an Oerlikon anti-aircraft gun, four spouts of 35-millimeter bullets, turned around, while firing, at random. A computer bug would be responsible.
- Venus's mission: move to 5,000,000 km from the planet, instead of the planned 5,000 km. Cause: replacement of a comma by a point (in US number format).
- Mariner 1: The first space probe of the Mariner program, sent by NASA on July 27, 1962. The probe was destroyed shortly after taking off. Cost: $80 million. Cause: A forgotten hyphen in a Fortran program.

It is critical to investigate software reliability in order to avoid such a catastrophe. This necessitates the use of well-defined design and validation methodologies in the testing process. We're all set to go over the fundamentals of software testing.

1.2. Test definition

According to the IEEE (Standard Glossary of Software Engineering Terminology). Testing is the automatic or manual performance or assessment of a system or component to ensure that it satisfies its requirements or to find variations between expected and actual outcomes.

1.3. Different Kinds of Testing

There are many kinds of testing, but one way to classify various testing techniques is shown in Figure 1.1.

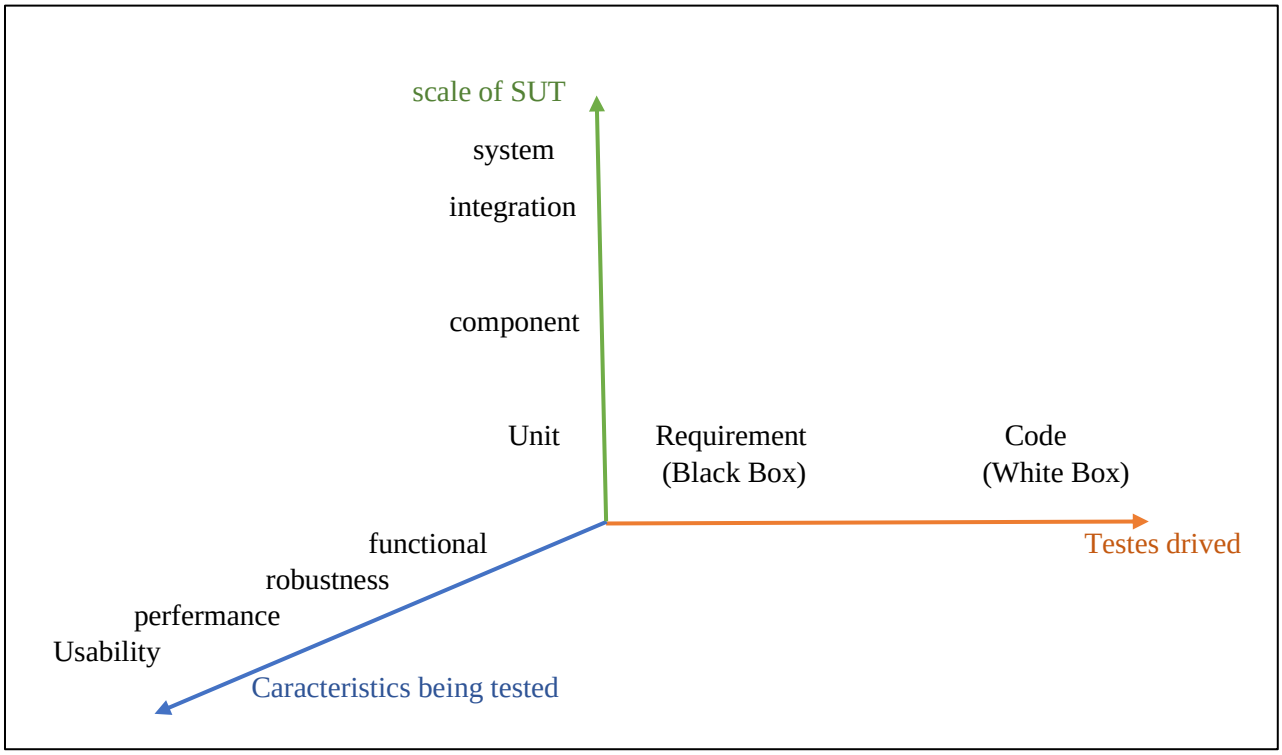


Figure 1.1: different kinds of testing

These different kinds of testing are defined in three dimensions.

1. The first dimension is showing the scale of SUT, which ranges from a small unit up to whole system.
2. The second dimension shows the different characteristics that we may want to test
3. The third dimension shows the kind of information, we may want to use during software testing.

Various kinds of testing, shown in Figure1.1, are described below.

1.3.1. Unit testing

Unit testing is a technique for ensuring that a specific section of code, such as a single function or a single class, works properly. These tests are frequently carried out by programmers themselves in order to ensure that a certain function behaves as expected.

1.3.2. Component Testing

A component is made up of multiple pieces that are strongly connected to one another. During component testing, each component/subsystem is examined separately to ensure that it is functioning properly.

1.3.3. Integration Testing

An integration test is carried out to see how well many components operate together. It's used to test and verify the interfaces of different components that are integrated with one another. In other words, it's used to identify any discrepancies that may exist between all of the components that are linked together.

1.3.4. System Testing

System testing is a type of testing that is carried out on a fully integrated piece of software or hardware. It is a type of black box testing that doesn't require any understanding of the code's or logic's inner workings [1].

1.3.5. Functional Testing

One of the most frequent forms of testing is functional testing, which confirms that the system behaves appropriately and meets at least part of the requirements, models, or other design paradigms used to describe the application. It's also known as "behavioural testing" and is used to identify flaws in a system's functionality.

1.3.6. Robustness Testing

Robustness testing is performed to discover system problems under erroneous settings. For example, by introducing unexpected inputs or testing the system independently of its dependent applications. A system is often regarded to be robust if it does not hang or crash during testing.

1.3.7. Performance Testing

A performance test is carried out on a system to determine its throughput under high load. The purpose of this test is to see if the system crashes when there aren't enough computing resources.

1.3.8. Usability Testing

Usability testing evaluates a product by identifying user interface issues that make the program difficult to use or cause the user to misinterpret the software's results. During this test phase, extra soft qualities such as ease of use and product appearance and feel can be evaluated. This is an essential usability technique since it provides immediate feedback on how real users interact with the system [2].

1.3.9. Black Box Testing

A SUT is treated as a Black Box in black box testing, which means we have no knowledge of the system's internal structure.

The tests in black box testing are created from the system requirements, which characterize the system's intended external behavior. As a result, black box testing offers the benefit of "an unbiased view" on one hand, but the problem of "blind probing" on the other [3].

1.3.10. The White Box Testing

The tests in white box testing are created utilizing the system's implementation code. For example, a set of tests might be created to guarantee that every statement or function in the code is covered.

As a result, each test case will run each statement or function. White box testing techniques may also be used to assess the completeness of a test suite developed using black box testing techniques. This enables the software team to study sections of a system that are rarely tested while also ensuring that the most critical function points are examined [4].

1.4. Model-Based Testing

The methods and strategies for automatically deriving abstract test cases from abstract formal models, generating concrete tests from abstract tests, and manually or automatically executing the resultant concrete test cases are referred to as model-based testing. The entire idea/concept underlying MBT is centered on the design and testing of a model of a System Under Test [5].

Model-based testing has a number of benefits, including a high level of automation, the capacity to produce large quantities of non-repetitive useful tests, the ability to analyze regression test suites, and the ability to estimate a variety of statistical measures of software quality. Model-based testing is commonly thought of as a type of black-box testing [6]. Therefore, scope of model-based testing is shown in Figure 1.2.

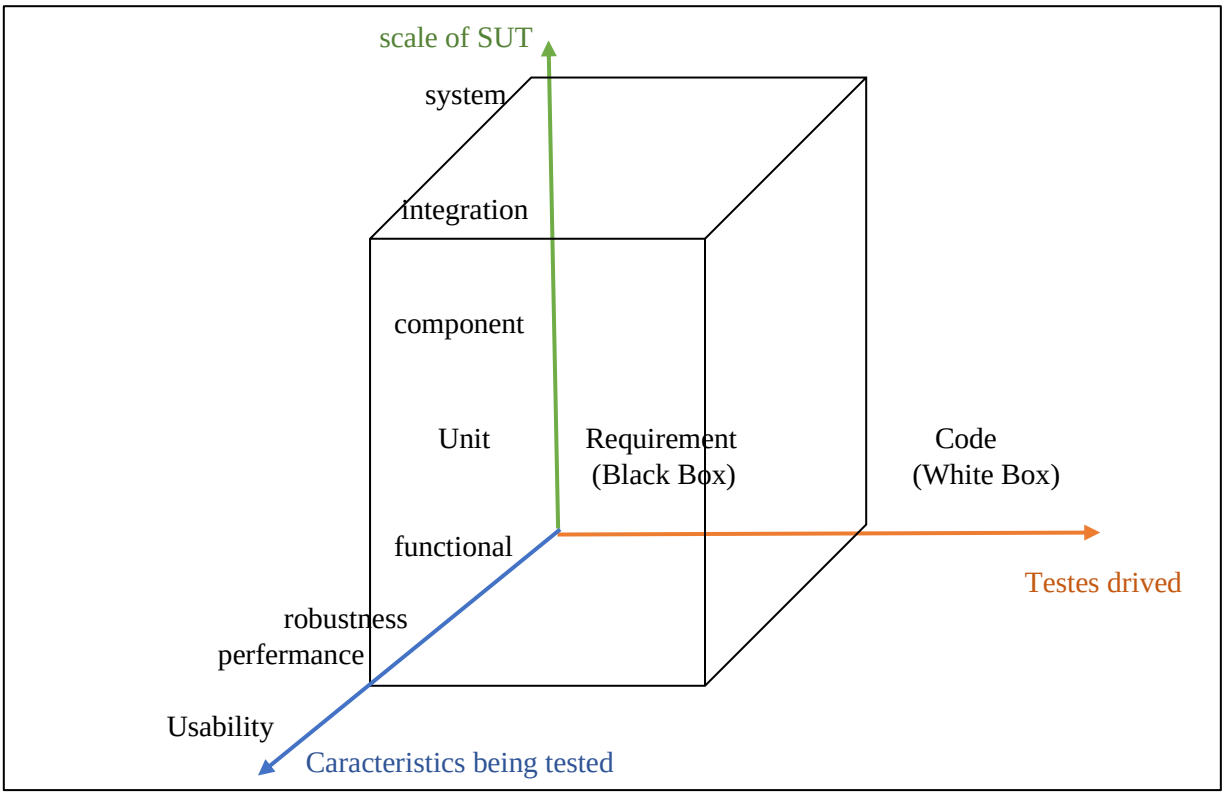


Figure 1.2: MBT in testing process

As model-based technique is usually used for black box testing, therefore the scope of model-based testing includes only the requirements of system model (see Figure 1.2). The main use of model-based testing is to generate functional testing, which covers the complete scale of SUT (see Figure 1.2). However, model-based technique can also be used for robustness testing, in which invalid inputs can be given to the SUT. It is not yet widely used for performance testing, but this is an area under development [7]. The following are the four main approaches known as model-based testing [7]:

1.4.1. Generation of test input data from a domain model

The model provides the information about the domain of the input values. The test generation involves clever selection and combinations of a subset of those values to produce test input data. This approach is obviously of great practical importance, but it does not solve the complete test design problem because it cannot provide any information, that either the test case passed or failed.

1.4.2. Generation of test cases from an environment model

This approach uses a model to describe the expected environment of the SUT. From these environments, sequence calls can be generated from this model, but generated sequence calls do not specify the expected output of the SUT. The environment model does not model the behaviour of the

SUT, meaning that it is not possible to predict the output values. In other words, it is difficult to determine accurately, that either a given test passed or failed.

1.4.3. Generation of test cases with oracles from a behaviour model

The third meaning of model-based testing is the generation of executable test cases which include oracle information or some automated check on the actual output values to see if they are correct. Oracle information is input values associated with operations and the corresponding expected output values. This is a more challenging task than the two previously mentioned approaches. The test generator must know enough about the expected behaviour of the SUT, such as the relationship between input and output, in order to generate test cases with oracles. Hence, the model must describe the expected behaviour of the SUT. Thus, this is one of the four approaches, described in this Section, which addresses the whole test design problem from choosing input values and generating sequence calls to generate executable test cases that include verdict information.

1.4.4. Generation of test scripts from abstract tests

The final approach assumes an abstract description of a test case, such as a UML sequence diagram, and focuses on transforming that abstract test case into a low-level test script that is executable. The model is the information about the structure and API of the SUT, and the details of how to transform a high-level call into executable test scripts.

1.5. Model Based Testing Process

The automation of the design of black box tests is termed as model-based testing [7]. The distinction from traditional black box testing is that instead of manually developing tests based on requirement documents, a model is constructed that acts like an expected SUT. A general process of model-based testing is shown in Figure 1.3.

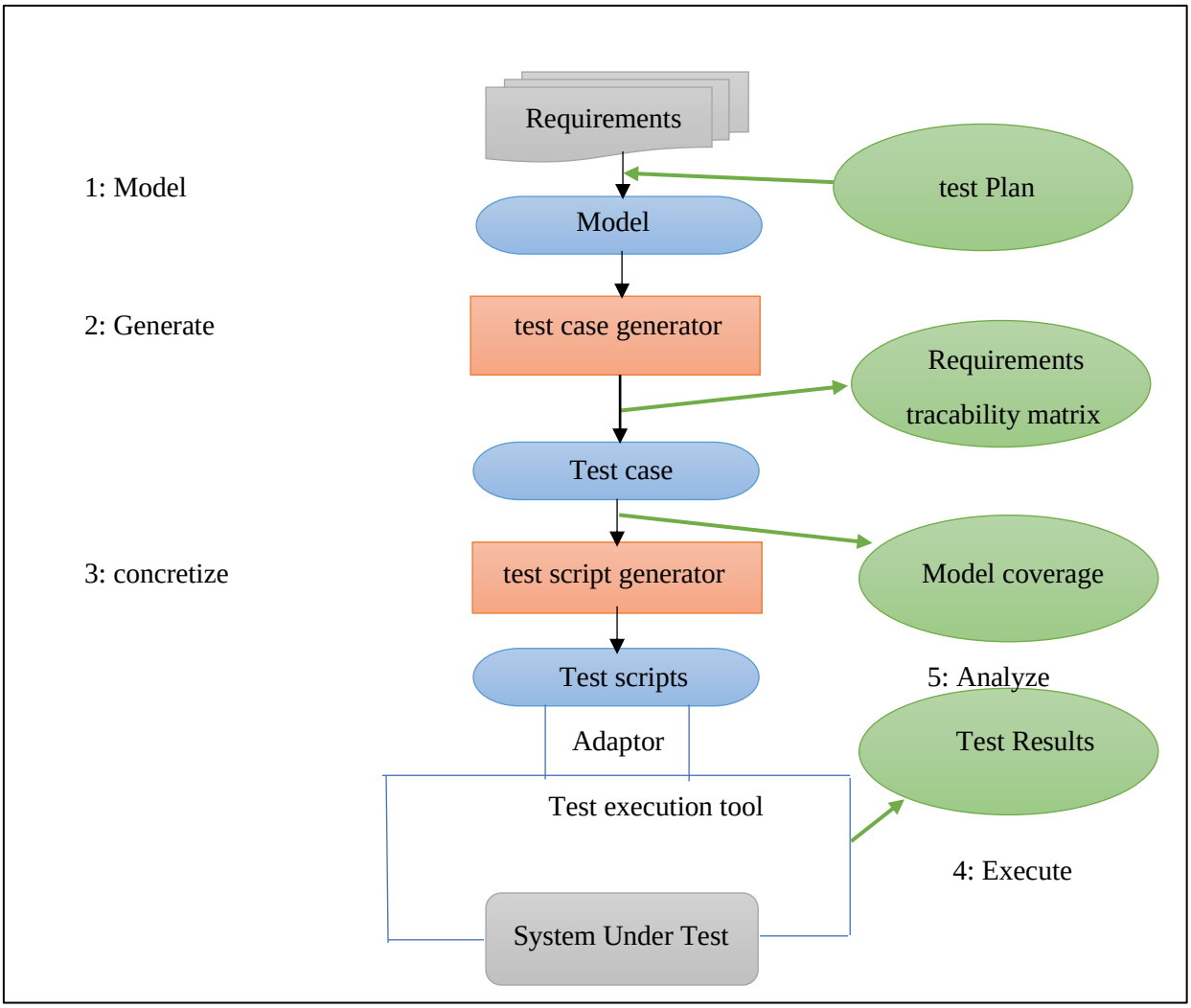


Figure 1.3: MBT process

Model-based testing automates the creation of the traceability matrix and the precise design of test cases. The test designer builds an abstract representation of the system under test rather than creating hundreds of test cases. The model-based testing tool is then used to produce a collection of test cases based on the model, resulting in a reduction in design time. Using diverse test selection criteria, a range of test suites may be built from the same model [7].

The process of model-based testing can be divided into five main steps:

1.5.1. Model the SUT and/or its environment

The first step of model-based testing is to construct an abstract model, which could be really a simplified, behavioural model of the system being tested. It should not be very detailed, but it should focus on the key aspects to be tested and be based on the specified requirements [7].

1.5.2. Generate abstract tests from the model

After creating the behavioural model of the system, the next step is to generate abstract test cases from that model. Hence to generate the test cases, a test selection criterion needs to be specified, since the number of possible tests may be infinite. For example, interaction with the test generation tool might be necessary to focus on a particular part of the model or to choose a particular model coverage criterion, such as to cover all transitions or cover all states in a finite state machine [7].

1.5.3. Concretize the abstract tests to make them executable

Since the model is an abstraction of the SUT, therefore the abstract test cases cannot be executed directly. A requirement traceability matrix, which links functional requirements with test cases to determine which requirements are covered by an individual test case, or various coverage reports are additional outputs of this step for most model-based testing tools. Coverage reports indicate how well the test cases cover the behaviour of the model, while a requirement traceability matrix traces the link between functional requirements and generated test cases [7].

When the abstract test cases are generated, they need to be transformed into executable concrete tests. This may be done by some separate transformation tool or it could be done by writing some adaptor code that implements each abstract operation to map against the lower level SUT interface. The aim of this step is to bridge the gap between abstract test cases with the concrete SUT by adding details not included in the abstract model [7].

This two-layer approach, i.e., abstract tests and concrete test scripts, has the advantage of being independent of the language used to write tests and of the test environment. Hence, just by changing the adaptor code, the tests can be reused in different test execution environments [7].

1.5.4. Execute the tests on the SUT and assign verdicts

The executable test scripts are then executed against the SUT. Using online model-based testing, the tests will be executed as they are produced. In this case the model-based testing tool handles execution and recording of the results. With offline model-based testing, a set of concrete test scripts has been produced. Hence the existing test execution tools and practices can be used [7].

1.5.5. Analyze the test results

Finally, the test execution results are analyzed, and correct actions have to be taken in order to fix the bugs. For each failed test it must be determined what caused the failure. It might be due to a fault in the SUT or to a fault in the test case itself. In the latter case this must be due to a fault in the adaptor code or in the behavioural model of the system. Hence, feedback about the correctness of the model is given in the last step [7].

Model-based testing is good at discovering SUT faults, but it's also good at revealing requirements flaws even before running a single test, according to experience.

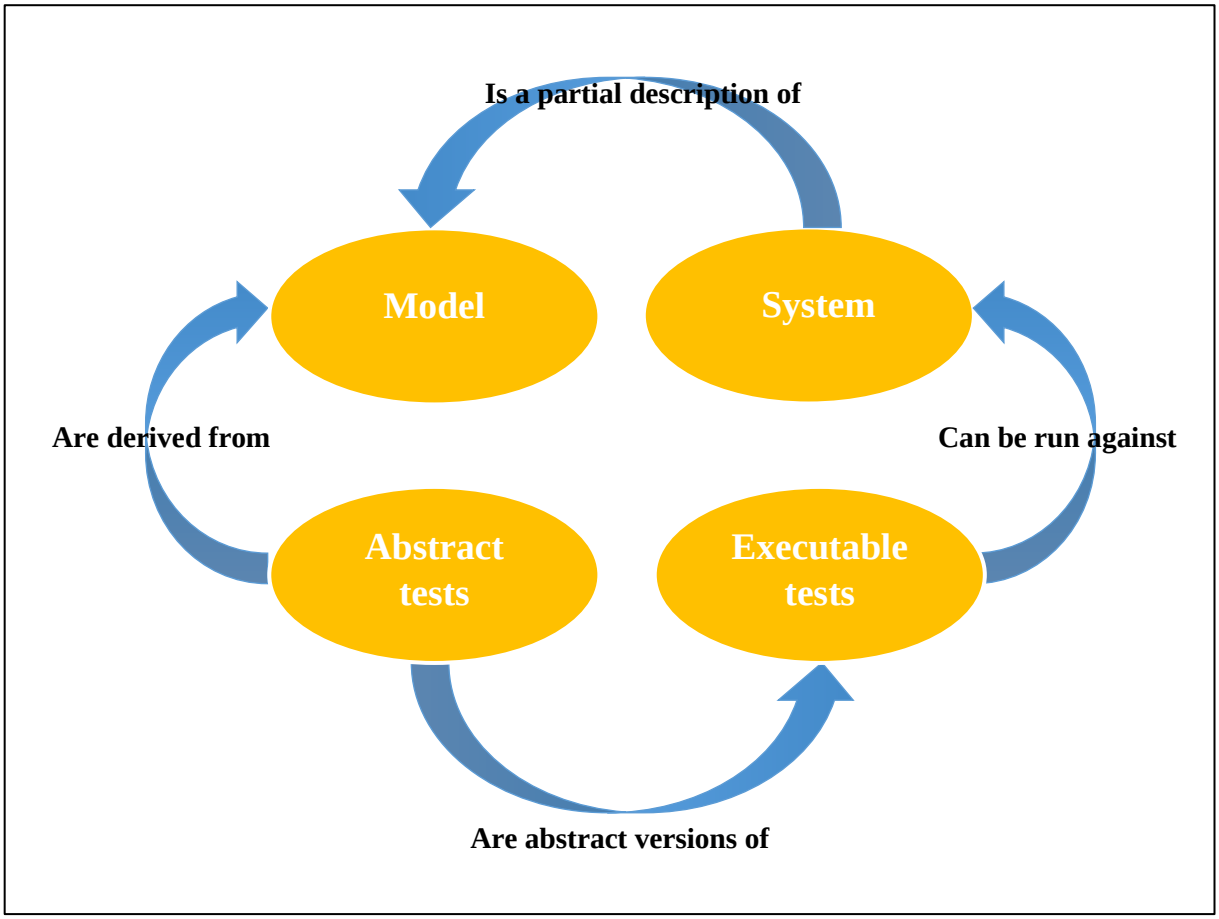


Figure 1.4: MBT principle

1.6. Classification of MBT Techniques

MBT is frequently broadened to include techniques that just employ a graphical language to express test cases, such as UML sequence diagrams. MBT is defined as a method for generating executable tests from a formal model of the system under test (SUT) utilizing a range of test selection criteria in this document. There are three sorts of models that are commonly used: Test models (that reflect the requirements on the system by modeling the behavior of a potential user), Design models (that specify the intended behavior of the system as it is implemented), Models constructed from the source code directly [8]. the source code directly [8].

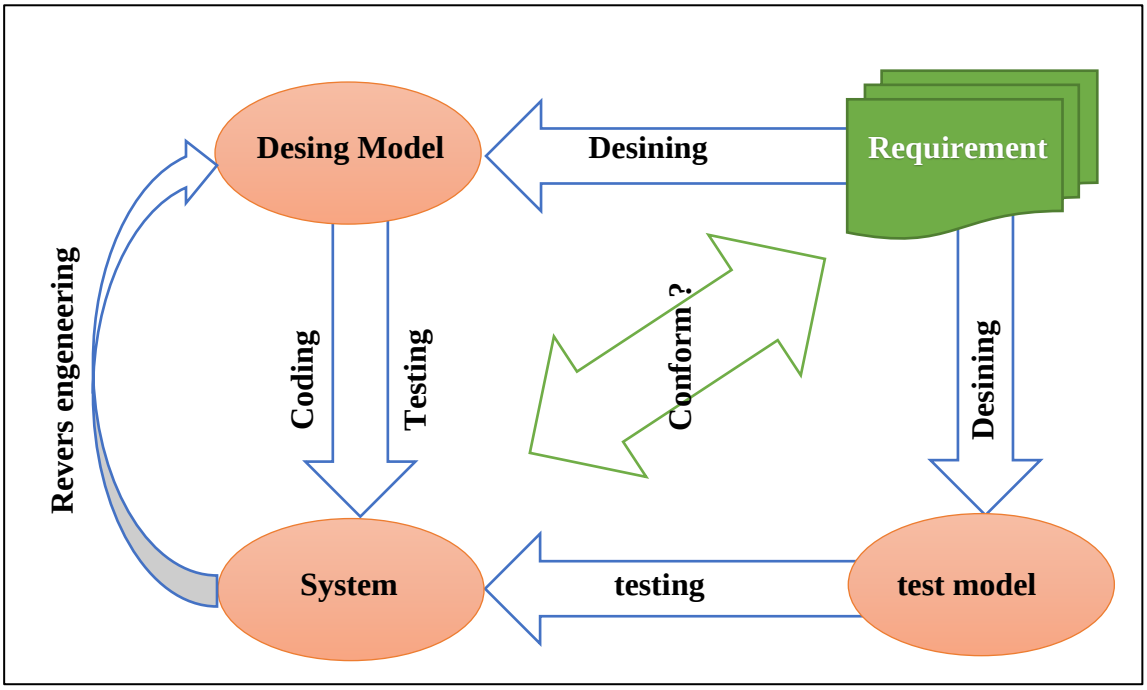


Figure 1.5: the design and test process.

The test derivation approaches applied on the different model types are typically classified into specification-based testing (or black-box testing) if they are based on requirement models and program-based testing (or white-box testing) if the source code is used as the underlying model.

Here a different classification scheme is applied that refers to the used test selection criteria: (a) coverage-based, (b) fault-based, and (c) mutation-based approaches (Table 1.1) [8].

	Coverage-based	Fault-based	Mutation-based
Test models	X		
Desing models	X	X	X
Source code models	X		X

Table 1.1: MBT approaches applied to different model types.

The underlying model is interpreted as a (directed) graph structure with a collection of vertices and edges in coverage-based techniques. So, the efficacy of syntactical coverage criteria in terms of error detection power has been supported by practical investigations. For example, a test case must cover all vertices in a graph or all pathways through a digraph (only).

Apart from the unrealistic all-path requirement, there is no "optimal" coverage criterion. Coverage-based test derivation procedures are applicable to any of the models discussed above due to their abstract interpretation. Fault-based techniques try to create tests based on a fault model that is imposed on the

system's design model. The W and UIO methods, as well as their offspring and modifications, are well-known methodologies.

A finite set of exhaustive tests of a given fault model and additional assumptions on the design model and the SUT may be derived, implying that these tests can discover any defect in the SUT provided its error class has been defined in the fault model. While fault-based test techniques are extremely effective and have a lengthy research history, they are limited to a few application areas, such as communication protocol testing. As a result, these methods are rarely used in industrial projects. Finally, mutation-based testing methodologies take use of the notion that a test with a strong error detection capability should be able to distinguish a large number of modified models (i.e., incorrect models) from the correct model [8].

A mutant is a syntactic mutation of the right model, such as the digraph's vertices being modified or added. Because the number of mutants created for a particular model is often unlimited, the choice of tests with a high error detection capability is largely determined by the size of the mutants generated.

In terms of current practical application, MBT techniques based on test models and proper coverage requirements reign supreme, in addition to being simple to implement using graphical specification techniques such as UML and the UML 2.0 Testing Profile in particular, these approaches benefit from existing test-driven development practices [8].

They also get around the test oracle issue that comes from deriving tests from design models. It may be difficult to disagree on the objective of derived tests if production code and test cases are created automatically from the same model.

However, breakthroughs in fault-based testing methodologies are predicted in the long term since they create tests of very high quality, but they are now impeded by the lack of adequate design models for most application domains. In certain specialized applications, mutation-based testing is predicted to play a role [8].

1.7. Benefits of Model Based Testing

In this Section, the benefits of model-based testing will be discussed. The most attracting benefit of model-based testing is that it automatically generates interesting cases from the system model.

The other major benefits, due to which one can use model-based testing technique are: SUT fault detection, reduced testing time and cost, improved test quality, requirement defect detection, traceability and requirement evolution.

1.7.1. SUT fault detection

The aim of model-based testing is the exposure of the failures in the SUT, which usually caused by exhaustive combinations of inputs, memory leakage, and sometime failures occur due to exercising different combinations of variables. Comparative studies [11] [13] [10] [9] show that model-based testing works better at fault detection than manually designed tests.

However, its fault detection power depends on the skill and experience of those writing the model and choosing the test selection criteria [7].

1.7.2. Reduced testing time and cost Model based testing practices

will lead to less time and effort spent on testing in case of time needed to write and maintain the model, as well as the time spent on directing the test generation is less than the cost of manually designing and maintaining a test suite. It might also save time during the failure analysis stage after test execution.

Firstly, because failures are reported in a consistent way and secondly, because some model-based tools are capable of finding to shortest possible test sequence that causes the failure. Thirdly, since not only the code can be inspected, but also the abstracted test cases which give an overview over the test sequence through the model [7].

1.7.3. Improved test quality While performing manual testing

The quality of tests is highly dependent on the test engineer, and the test design process is usually not reproducible. Model-based testing, however, uses an automated test generator based on algorithms and heuristics to choose the test cases from the model, which makes the design process systematic and repeatable. Since the input data and the test oracles are generated from the model, the cost of generating more executable test scripts is just the computing time required to generate them [7].

Model Coverage The test progress and the generated test cases can be evaluated using a coverage criterion, defined before the test generation. Coverage can also be expressed for a model. Therefore, model coverage is another heuristic that provides insight into the thoroughness and effectiveness of the testing effort, especially when testing does not reveal failures. Coverage typically deals with the control-flow through the model [7]. [13] Defect Detection Requirements Typically, while writing the model for testing, it exposes issues in the informal requirements.

As in model-based testing, the first step is to create an abstract model of the SUT, which usually exposes requirements issues, this is a major benefit of model-based testing because requirements problems are a major source of system problems [7].

1.7.4. Traceability

Is the ability to relate each test case to the model, to the test selection criteria, and even to the informal system requirements.

Traceability helps to explain the test case as well as gives the justification for why it was generated. Furthermore, it can be used to optimize test execution as the model evolves, since it enables the possibility to execute just the subset of the tests that are affected by the model modifications. From an abstract view, traceability is a relation between the elements of the model and the test cases [12]. Requirements evolution A considerable amount of effort is often required to update the test suite as the requirements of the system change while performing manual testing.

Model-based testing, on the other hand, just the model has to be modified, and the tests may be regenerated. Because the model is generally considerably smaller than the test suite, upgrading the model saves time compared to manually updating all tests, allowing for faster reaction to changing needs.

Model Based Testing's Drawbacks Model-based testing has a number of advantages, as previously stated, but it also has certain drawbacks. Even with a huge test set, model-based testing cannot ensure that all discrepancies between the model and the implementation will be found. This, however, is a restriction that applies to all types of testing.

The following are some of the drawbacks of model-based testing. criteria that are no longer valid Informal requirements might become out of date as a software project progresses. If this is the case while employing model-based testing, the incorrect model will be constructed, and test case execution will result in a large number of errors in the SUT.

Metrics that are useless A number of test cases are frequently developed as gauges of how the testing is progressing during the manual test design process. When using model-based testing, such measurements aren't relevant because the method can yield a large number of test cases. Other metrics, such as SUT code coverage, requirements coverage, and model coverage metrics, should be used instead of test progress measures.

Model-based testing is being used incorrectly. Some components of the SUT may be difficult to simulate in software testing, and these parts may need to be manually tested. It is not required for model-based testing to be applicable to all domains of software. The concern is that knowing which features of the SUT should be modelled and which should be tested manually or using other tools or techniques requires some expertise with model-based testing.

Tester abilities A practical disadvantage of model-based testing is that it necessitates a different set of skills than manual test design. In addition to being specialists in the application domain, model

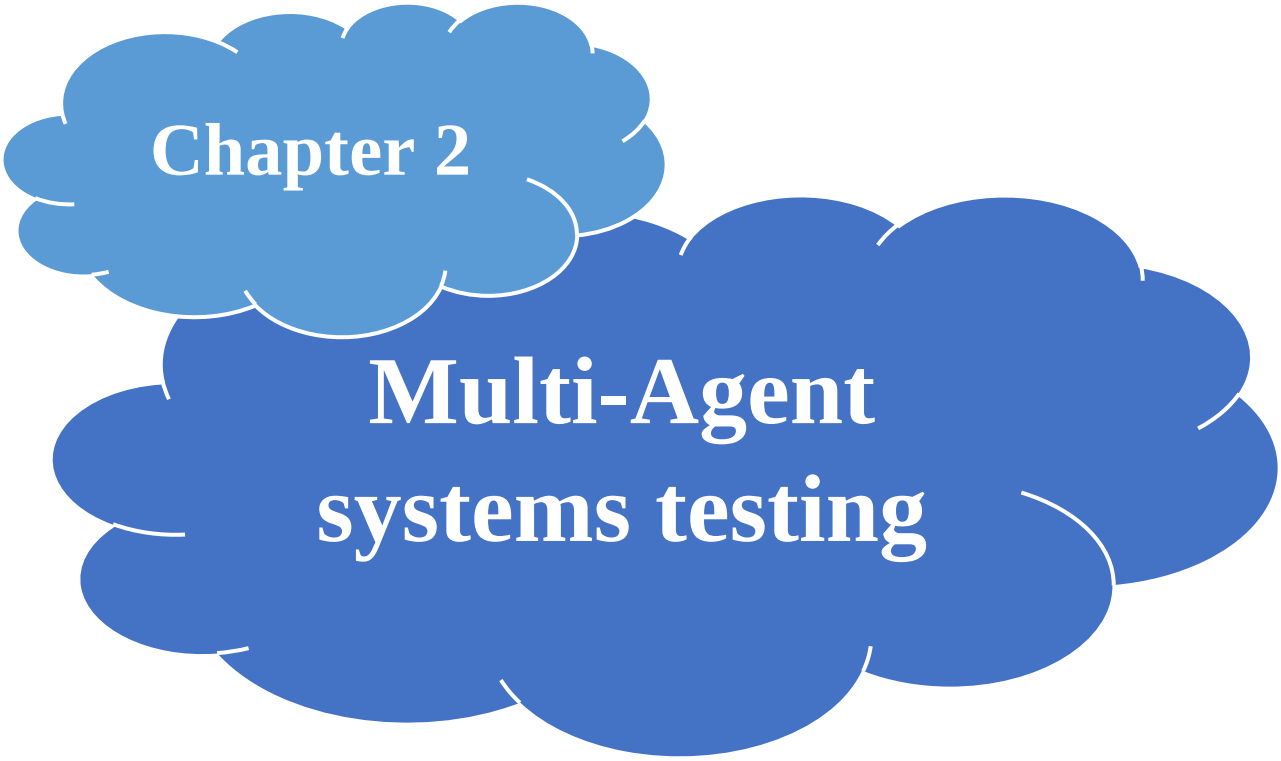
designers must be able to abstract and construct models. When doing model-based testing for the first time, this necessitates training expenditures and a steep learning curve.

State space explosion Some drawbacks of model-based testing cannot be avoided completely. For state models the most prominent problem is state space explosion. Models of any non-trivial software functionality can grow beyond manageable levels. Almost all other model-based tasks, such as model maintenance, checking and reviewing, non-random test generation and achieving coverage criteria, are affected in this scenario [12].

It's time to examine the results of failed tests. If any of the produced tests fails, it must be determined if the failure is due to a problem with the SUT, the adapter code, or a model issue. This is analogous to manual testing, in which it is necessary to determine if the failure was caused by a defect in the SUT or in the test script. Model-based testing, on the other hand, provides test sequences that are more complicated and less intuitive than those created manually. As a result, determining the cause of a failed test may be more complicated and time-consuming.

1.8. Conclusion

Despite the formal problems faced by testers, testing is now an important aspect in improving the quality of software. There is now no method capable of demonstrating a program's entire correctness. It is, in fact, a vocation in and of itself. It is the sole activity in the development cycle where you may view all of a software product's capabilities, according to the industry. It is the most expensive stage since it accounts for 30 to 40% of software development expenditures, depending on the criticality of the project (usually 1/3 of the schedule). The notion of multi-agent system testing will be introduced in the following chapter.


Chapter 2
Multi-Agent
systems testing

2. MULTI-AGENT SYSTEMS TESTING

2.1. Introduction

Multi-agent systems are part of the artificial intelligence discipline, and they are systems that are seen in a different way than traditional computer engineering. Where traditional computer problem solving has reached its limits, multi-agent systems come into play. Unfortunately, given the opportunities it provides in many domains such as social sciences, computer sciences, experimental sciences, and industry, this field is underutilized today. However, the limitations of contemporary computer and industrial systems mean that developing multi-agent systems to satisfy the expanding requirements of many more conventional areas, whether in terms of time, efficiency, or production, becomes possible and even attractive.

2.2. Definition

As described thus far, Agent-Oriented Software Engineering (AOSE) techniques primarily provide disciplined approaches to evaluate, develop, and implement MASs. However, how multi-agent systems may be examined has received little attention. Only a few of these approaches include a formal verification procedure. To allow automatic verification of inter-agent communications, the MaSE and MASCommonKADs approaches provide a verification step based on model checking.

Desire proposes a verification phase based on mathematical proofs, with the goal of proving that a system adheres to a set of properties under a given set of assumptions. Only a few iterative approaches provide incremental testing techniques and tools. PASSI/Agile PASSI, AGILE are two examples. The JUnit test framework is used by AGILE to define a testing phase. They needed to implement a sequential agent platform, which simulates asynchronous message-passing, in order to use this tool, which was designed for OO testing, in a MAS testing context.

Having to execute unit tests in an environment different from the production environment results in a set of tests that does not explore the hidden places for failures caused by the timing conditions inherent in real asynchronous applications. Agile PASSI [14] proposes a framework to support tests of single agents.

Despite proposing valuable ideas concerning MAS potential levels of tests, PASSI testing approach is poorly documented and does not offer techniques to help developers in the low-level design of unit test cases. Hence, we can say that few research works have been undertaken in order to provide MASs developers with a detailed testing process and valuable tools to support testing activities.

2.3. A unit test approach for multi agent systems

Our testing approach calls attention to the test of the smallest building blocks of the MAS: the agents. Its basic idea is to verify whether each agent in isolation respects its specifications under normal and abnormal conditions. There are different proposals for representing an agent. Our approach is based in the definition detailed in [17] and presented below:

An agent is an autonomous, adaptive and interactive element that has a mental state. The mental state of an agent is comprised by: beliefs, goals, plans and actions. Every agent of the MAS plays at least one role in an organization. One of the attributes of a role is a number of protocols, which define the way that it can interact with other roles. Agents encapsulate a very complex internal structure (often composed of several classes and/or methods). In order to verify whether these inner components contain faults, we can use traditional unit testing techniques.

However, the agent is the unit of modularity of MASs - which is internally coherent and has minimum coupling with the rest of the system – and as such should be tested as a whole.

A running MAS is a web of agents that interact with each other and with an external environment? Since, this kind of interaction differs in nature from the direct method call that takes place within an agent (among the classes that constitutes it) we need to devise specific techniques to test each individual agent.

Figure 2.1 depicts a test of agent A, which needs a service provided by an agent that plays role B. In this figure, A is the Agent Under Test (AUT). Along this paper, we will use this term to refer to the agent being tested.

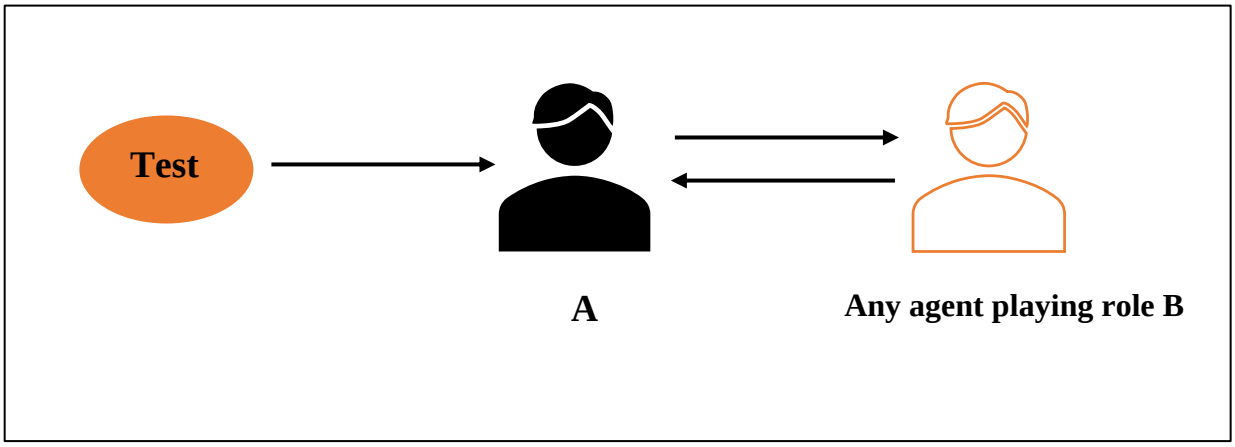


Figure 2.1: Unit Testing Agent A

In order to test A in isolation, a valuable strategy is to define a "dummy" version of B, usually called stub. Stubs are fake implementations of production code that return canned results.

Makinnon et al proposed the Mock Object test design pattern [18], and since then, Mock Objects have been recognized as a useful approach to the unit test and design of object-oriented software.

Mock object It is a regular object that acts as a stub but also includes assertions to instrument the interactions of the target object with its neighbors. Mock objects can be used in MAS testing to simulate real environmental resources under the conditions defined by [19].

In our approach, we adapted Mackinnon et al. idea to MAS testing context and defined the concept of: Mock Agent. A Mock Agent is a regular agent that communicates with just one agent: the AUT. It has just one plan to test the AUT. The Mock Agent's plan is equivalent to a test script, since it defines the messages that should be sent to the AUT and the messages that should be received from it. By testing an agent in isolation using Mock Agents the programmer is forced to consider the agent's interactions with its collaborators (or competitors), possibly before those collaborators (or competitors) exist.

The next section details our unit test approach, which uses Mock Agents to guide the design of each test case. Approach's Overview Figure 2.2 depicts our agent unit test approach that is composed of five participants:

- Test Suite: which consists in a set of Test Cases and a set of operations performed to prepare the test environment before a Test Case starts.
- Test Case: defines a scenario – a set of conditions – to which an Agent Under Test is exposed, and verifies whether this agent obeys its specification under such conditions.
- Agent Under Test (AUT): is the agent whose behavior is verified by a Test Case.
- Mock Agent: consists in a fake implementation of a real agent that interacts with the AUT. Its purpose is to simulate a real agent strictly for testing the AUT.
- Agent Monitor: is responsible for monitoring agents' life cycle in order to notify the Test-Case about agents' states.

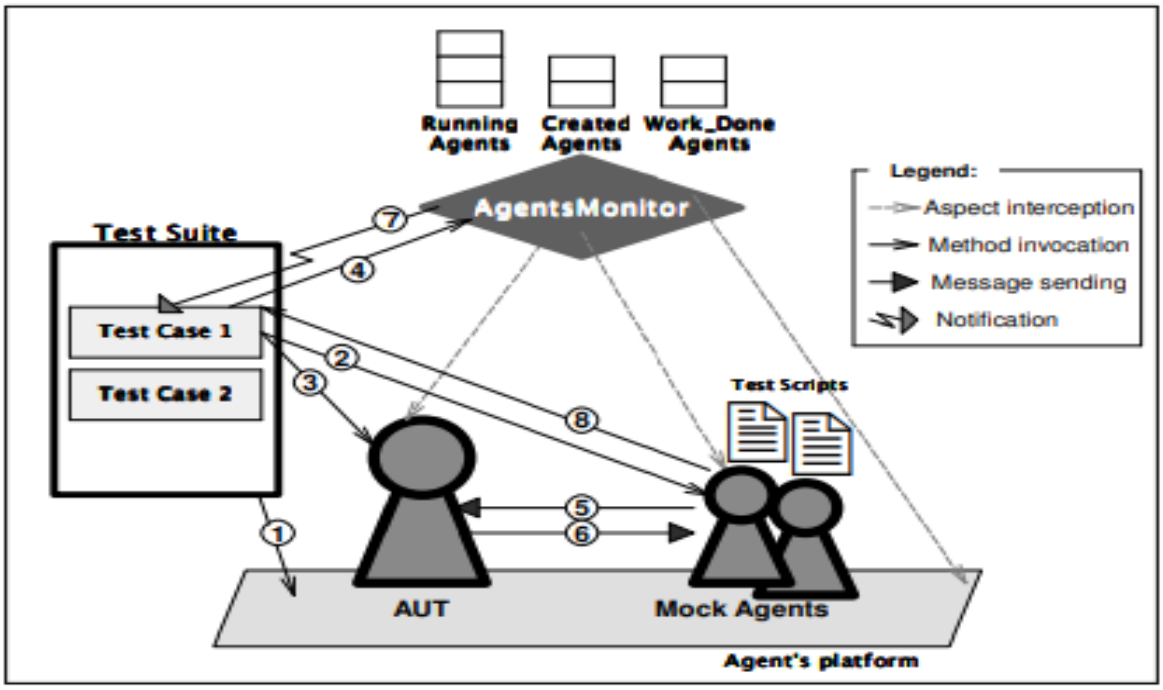


Figure 2.2: Workflow between the participants of a unit test.

The common structure represented in Figure 2.2 is followed by each agent unit test. The Test Suite develops the agent's platform as well as any other elements required to set up the test environment in step one, following that, a Test Case is created.

In the scenario provided by the Test Case, the Test Case produces Mock Agents for every role that interacts with the Agent Under Test (step 2), It then generates the Agent Under Test (step 3)2 and notifies the Agent Monitor when the interaction between the AUT and the Mock Agents is complete (step 4).

The AUT and the Mock Agent begin to communicate at this moment. The Mock Agent delivers a message to the AUT, which responds (steps 5 and 6) alternatively the AUT responds to the Mock Agent (steps 5 and 6). They can execute the test specified in the Mock Agent's plan as many times as necessary by repeating steps 5 and 6. (for instance, the Mock Agent can reply three messages before finalizing its test activity).

The Agent Monitor maintains track of changes in the agents' life cycle during the whole interaction process, to do this, it employs three lists, as seen in Figure 2.2:

2.3.1. Created Agents List

Stores the IDs of the agents that have been created but are not yet operating - an ID is any piece of information that identifies an agent uniquely.

2.3.2. Running Agents List

Keeps track of the running agents' IDs.

2.3.3. WorkDone Agents List

Maintains IDs of the Mock Agents that have completed their plan (equivalent to a test script). When a Mock Agent completes its plan, the Agent Monitor adds the Mock Agent's ID to the WorkDone list and informs the Test Case that the Mock Agent's contact with the AUT is complete (step 7). Lastly, the Test Case asks the Mock Agent whether or not AUT acted as expected (step 8). This agent unit testing approach has two main concerns:

1. The creation of a Test Case based on the employment of Mock Agents;
2. And the execution of the Test Case, which relies on the Agent Monitor to indicate when the test script (as specified in the Mock Agent's plan) has been completed.

Next Sections will detail these two concerns. Test-Case Design based on Mock Agents A very important consideration in program testing is the design and creation of effective test cases [20].

Testing, however creative and seemingly complete, cannot guarantee the absence of all errors [20]. Test-case design is so important because complete testing is almost impossible; a test of any non-trivial program must be necessarily incomplete.

The obvious strategy, then, is to try to make tests as complete as possible. Given constraints on time and cost, the key issue of testing becomes: What subset of all possible test cases has the highest probability of detecting the most errors? The study of test-case design methodologies supplies answers to this question [20]. In general, the least effective methodology of all is to arbitrarily choose a set of test cases. In terms of the likelihood of detecting the most errors, an arbitrarily selected collection of test cases has little chance of being an optimal, or even close to optimal, subset.

Unit test approaches for MASs proposed so far does not define a methodology for test-case selection. Below we present an error-guessing [20] test-case design technique.

The basic idea of an error-guessing technique is to enumerate a list of possible error-prone situations and then write test cases based on the list.

- The process is as follows: 1.
- For each agent to be tested 1.1.
- List the set of roles that it plays. 2.
- For each role played by the AUT 2.1.
- List the set of other roles that interacts. with it. 3.

- For each interacting role: 3.1 Implement in a Mock Agent a “plan” that. codifies a successful scenario. 3.2.
- List possible exceptional scenarios that the.
- Mock Agent can take part. 3.3.
- Implement in the Mock Agent an extra plan.

This technique should be applied for each agent of the MAS, or a subset of the agents responsible for the “core” functionalities of the MAS. Such a choice will be guided by the time and cost constraints previously mentioned. At the end of this process, a Mock Agent comprises a set of expected behaviors (under successful and exceptional scenarios) of an agent interacting with AUT.

In order to help developers in the definition of Mock Agents plans, useful sources of information are sequence diagrams3 and the specification of protocols that regulates the interaction between MAS roles. As each Mock Agent exercises just one role of the AUT, rather than the wide interface that comprises all the features provided it, we call this approach “Role Driven Unit Testing”.

However, the notion of a role, while supported in many AO methodologies, is not used by some of them. In case the unit test developer is not using a role-based methodology, the process described above should be adapted. Instead of identifying each step according to the agent's role, he/she should define each step according to the agents' plans. Thus, the first two steps would be:

- For each agent of the MAS, list the set of plans that it performs;
- For each plan performed by the AUT, list the set of other agents that interacts with it.

Although our approach can help MAS developers in unit test cases construction, it does not intend to be complete. This technique should be combined with other strategies. The reason for such combination it is that: each test-case-design technique contributes a specific set of useful test cases, but none of them by itself contributes a thorough set of test cases [20].

Test Case Execution According to our approach, the plan of a Mock Agent comprises the logic of the test. Each Test Case just starts the AUT and the corresponding Mock Agent(s) and waits for a notification from the Agent Monitor – informing that the interaction between the agents have finished – in order to ask the Mock Agent(s) whether or not the AUT acted as expected.

To keep track of the changes in agents’ life cycle, Agent Monitor needs to include and remove information from the three lists described previously: Running Agents, Created Agents and WorkDone Agents. In order to access such information, the Agent Monitor needs to observe specific application events, such as: agent creation, the moment at an agent starts running, agent finalization, and so on.

To prevent monitoring concern from becoming scattered across multiple platform modules and tangled with other application concerns, the Agent Monitor participant is built upon the facilities of Aspect Oriented Software Development (AOSD) [15, 16].

2.3.4. Aspect Oriented Software Development

AOSD has been proposed as a paradigm for improving separation of concerns in software design and implementation, it proposes a new abstraction, called Aspect, with new composition mechanisms which support the modularization of crosscutting concerns.

The aspect abstraction aims at encapsulating concerns that crosscut several system modules. Since the Agent Monitor deals with the "monitoring" concern, which has a crosscutting nature, it is represented as an Aspect in our approach.

2.4. MAS Testing Problems

Creating a systematic software agent and MAS testing process: As with Formal Tropos, AOSE techniques have focused mostly on requirement analysis, design, and implementation, with little emphasis paid to validation and verification.

There is still no systematic testing method to complement analysis and design. This is a critical issue because without comprehensive and systematic standards, the development cost in terms of effort and productivity might skyrocket.

- They have their own motives for being proactive, which may differ from a user's concrete expectation but are nonetheless acceptable.
- In various executions, the same test input might yield different results.
- Because agents collaborate with one another, they may run correctly on their own but erroneously in a community, or vice versa.
- Furthermore, because agents may be trained to learn, outcomes from future tests using the same test data may change.

As a result, identifying appropriate and successful testing procedures for software agents is a critical topic in agent development. [21]

2.5. MAS Testing Approaches

Work in testing software agents and MAS can be classified into different testing levels: unit, agent, integration, system, and acceptance. Here we use general terminologies rather than using specific ones used in the community like group, society. Group and society, as called elsewhere, are equivalent to integration and system, respectively. Testing in MAS consists of five levels, as proposed in [22] and [23]:

- Unit testing tests all units that make up an agent, including blocks of code, implementation of agent units like goals, plans, knowledge base, reasoning engine, rules specification, and so on make sure that they work as designed.

- Agent testing tests the integration of the different modules inside an agent; test agents' capabilities to fulfill their goals and to sense and effect the environment.

- Integration or Group testing tests the interaction of agents, communication protocol and semantics, interaction of agents with the environment, integration of agents with shared resources, regulations enforcement; Observe emergent properties, collective behaviors; make sure that a group of agents and environmental resources work correctly together.

- System or Society testing tests the MAS as a system running at the target operating environment; test the expected emergent and macroscopic properties of the system as a whole; test the quality properties that the intended system must reach, such as adaptation, openness, fault tolerance, performance.

- Acceptance testing tests the MAS in the customer's execution environment and verifies that it meets stakeholder goals, with the participation of stakeholders

The data in Table 2.1 give an overview of the most recent and active work in this domain. They are organized according to the classification introduced in the previous section. The third dimension, maturity, will be tagged to each work. In the following sections [21]:

- we first summarize the contributions of the works that focus mainly on a particular testing level, e.g., agent; hereafter are the works that tackle more than one testing level.

In addition:

- we also summarize some interesting work that do early validation of the agent and MAS design based on simulation techniques and generated agent skeletons. It is non-trivial to organize.

	Unit	Agent	Integration	System	Acceptance
Active	Zhang et al. (2007, 2008, 2009), Tiryaki et al. (2006), Ekinci et al. (2008) , Nguyen et al. (2010)	Núñez et al. (2005), Coelho et al. (2006), Tiryaki et al. (2006), Gómez-Sanz et al. (2009), Nguyen et al. (2008, 2009, 2010)	Gómez-Sanz et al(2009), Nguyen et al. (2010)	Nguyen et al. (2010)	Nguyen et al. (2010)
passive		Lam and Barber (2005), Núñez et al. (2005), Gardelli et al. (2005), Fortino et al. (2006), Bernon et al. (2007), Cossentino et al. (2008)	Sierra et al. (2004), Botía et al. (2004), Rodrigues et al. (2005), Serrano and Bota (2009), Serrano et al. (2009), Sudeikat and Renz(2009)	De Wolf et al. (2005)	

Table 2.1: State-of-the-art on MAS testing

These works following the proposed classification since their subjects under test are agent design, not agent code.

However, we decide to include them because they are complementary to testing, and both types contribute to the final goal of ensuring the quality of the agent or MAS under development [21].

2.5.1. Unit Testing

Unit testing approach calls attention to the test of the smallest building blocks of the MAS: the agents. Its essential idea is to check if each agent in isolation respects its specifications under normal and abnormal conditions.

Unit testing needs to make sure that all units that are parts of an agent, like: (goals, plans, knowledge base, reasoning engine, rules specification, and even blocks of code work as designed), have to test the integration of the different modules inside an agent, test agents' capabilities to achieve their goals and to sense and effect the context. There are several works in agent testing level [21]:

2.5.1.1. Agile PASSI

Proposes a framework to support tests of single agents. They develop a test suite specifically for agent verification. Test plans are prepared before the coding phase in according with specifications.

Despite proposing valuable ideas concerning MAS potential levels of tests, PASSI testing approach is poorly documented and does not offer techniques to help developers in the low-level design of unit test cases.

2.5.1.2. Lam and Barber

Proposed a semi-automated process for comprehending software agent behaviors. The approach imitates what a human user (can be a tester) does in software comprehension and using it to verify and explain behaviors of agents at runtime.

Although the work did not deal with other problems in testing, like the generation and execution of test cases, the way it evaluates agent behaviors is interesting and relevant for testing software agents.

2.5.1.3. Nunez et al. S

Introduced a formal framework to specify the behavior of autonomous e-commerce agents. The desired behaviors of the agents under test are presented by means of a new formalism, called utility state machine that embodies users' preferences in its states, and there are two testing methodologies were proposed to check whether an implementation of a specified agent behaves as expected:

- In their active testing approach, they used for each agent under test a test (a special agent) that takes the formal specification of the agent to facilitate it to reach a specific state.
- The operational trace of the agent is then compared to the specification in order to detect faults. On the other hand, the authors.
- In their passive testing approach in which the agents under test were observed only, not stimulated like in active testing.

Invalid traces, if any, are then identified thanks to the formal specifications of the agents.

2.5.1.4. Coelho et al.

Proposed a framework for unit testing of MAS based on the use of Mock Agents. Even though they called it unit testing but their work focused on testing roles of agents at agent level according to our classification.

Mock agents that simulate real agents in communicating with the agent under test were implemented manually; each corresponds to one agent role.

2.5.1.5. Houhamdi

Introduces a suite test derivation approach for Agent testing that takes goal-oriented requirements analysis artifact as the core elements for test case derivation.

These test suites, on the one hand, can be used to refine goal analysis and to detect problems early in the development process. On the other hand, they are executed afterwards to test the achievement of the goals from which they were derived.

2.5.2. Integration Testing

Integration testing test the interaction of agents, communication protocol, interaction of agents, this approach exploits clustering techniques to build agent interaction graphs that support the detection of missed communication between agents that are expected to interact. Interaction protocol specifications corresponding to the conversation are fired and then analyzed to detect automatically erroneous conditions.

Information available in the specifications of these conventions gives rise to a number of types of assertions, such as time to live, role, cardinality, and so on. During test execution a special agent called Report Agent will observe events and messages in order to generate analysis report afterwards [21].

2.5.2.1. Ekinci et al.

They considered system goals as the source cause for integration and use them as driving criteria, apply the same approach for testing agent goals to test these goals and They define the concept of test goal. This concept represents the group of tests needed in order to check if the system goal is achieved correctly.

2.5.2.2. Nguyen et al.

Propose using ontologies extracted from MAS under test and a set of OCL constraints, which act as a test oracle, it's having as input a representation of the ontologies used, the idea is to construct an agent able to deliver messages whose content is inspired by these ontologies.

The resulting behaviors are regarded as correct using the input set of OCL constraints: if the message content satisfies the constraints, the message is correct. The procedure is support by eCAT, a software tool.

2.5.2.3. Houhamdi and Athamena

Introduced a novel approach for goal-oriented software integration testing. They propose a test suite derivation approach for integration testing that takes goal-oriented requirements analysis artifact for test case derivation and they have discussed how to derive test suites for integration test from architectural and detailed design of the system goals.

These test suites can be used to observe emergent properties resulting from agent interactions. This approach defines a structured and comprehensive integration test suite derivation process for engineering software agents by providing a systematic way of deriving test cases from goal analysis.

2.5.3. System and Acceptance Testing System testing tests

Test for quality properties, such as adaptation, openness, fault-tolerance, performance. At the system level of testing MAS, one has to test the MAS as a system running at the target operating environment; test the expected qualities that the intended system. Some initial effort has been devoting to the validation of macroscopic behaviors of MA.[21]

2.5.3.1. Sudeikat and Renz

Proposed to use the system dynamics modeling notions for the validation of MAS. These allow describing the intended, macroscopic observable behaviors that originate from structures of cyclic causalities.

System simulations are then used to measure system state values in order to examine whether causalities are observable.

2.5.3.2. Houhamdi and Athamena

Introduced a suite test derivation approach for system testing that takes goal-oriented requirements analysis artifact as the core elements for test case derivation.

The proposed process has been illustrated with respect to the Tropos development process. It provides systematic guidance to generate test suites from modeling artifacts produced along with the development process.

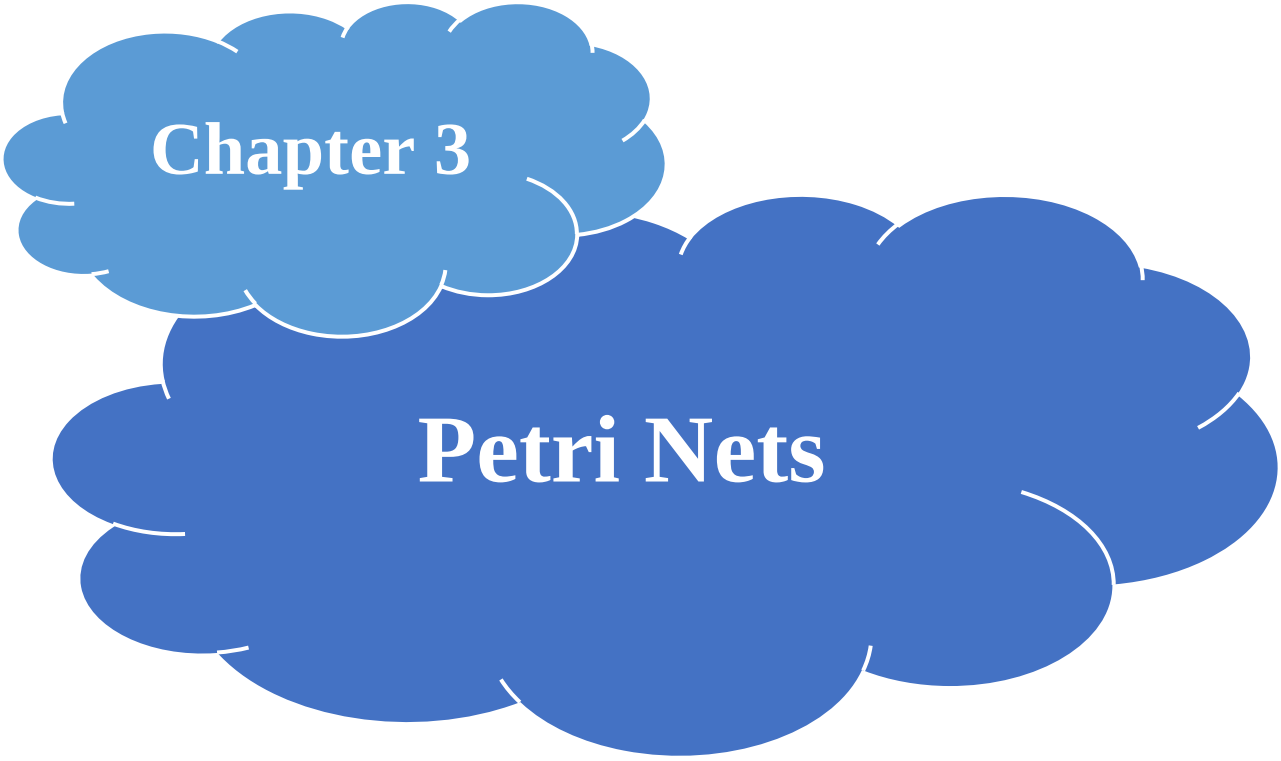
They have discussed how to derive test suites for system test from late requirement and architectural design. These test suites, on the one hand, can be used to refine goal analysis and to detect problems early in the development process. On the other hand, they are executed afterwards to test the achievement of the goals from which they were derived.

- Acceptance testing tests the MAS in the customer execution environment and verifies that it meets the stakeholder goals, with the participation of stakeholders.
- To the best of our knowledge, there is no work dealing explicitly with testing MAS at the acceptance level, agent, integration,
- System test harnesses can be reused in acceptance test, providing execution facilities.
- However, as testing objectives of acceptance test differ from those of the lower levels, evaluation metrics at this level, demand for further research.

2.6. Conclusion

We may infer from what we've seen in this chapter that testing is critical for software verification and repair. Because of the unique nature of the agents, current test methodologies are challenging to apply to multi-agent systems. The majority of extant MAS testing research focuses on the agent and integration levels.

The reference network, which is a specific petri network, is the basis for the test technique that will be detailed in this thesis.


Chapter 3
Petri Nets

3. PETRI NETS

3.1. Introduction

In recent years, formal approaches have become more popular, particularly in the creation of mission-critical software at various phases of the life cycle, such as analysis, design, and validation. Thus, by detecting and eliminating faults early in the development process while respecting the system's requirements and avoiding probable errors, these approaches make it feasible to acquire a very strong guarantee of the absence of defects in software.

There are a variety of formalisms (automata, temporal logic, Petri nets, and so on), but we were particularly interested in Petri nets because of their power, accuracy, and expressiveness. The following is an explanation of Petri nets.

Petri nets are a mathematical tool for modeling and verification of systems that, in addition to their analytical strength, provide a simple graphical representation that aids in the modeling of complex systems. As a result, the Petri net model is a graphical tool for modeling and analysis of systems that is well suited to the study of control structures. It enables sophisticated software to be controlled and ensured to be safe to use (aeronautics, transport industry, etc.).

3.2. Definition

Petri nets are a visual representation of a system in which numerous independent actions are taking place at the same time. Petri nets vary from finite state machines in that they may model numerous activities. There is always a single "current" state in a finite state machine that specifies which action can be performed next. There may be several states in Petri nets, each of which can evolve by altering the state of the Petri net. Alternatively, some, or perhaps all, of these states might evolve at the same time, resulting in many independent alterations to the Petri net.

The four components of a Petri net are Places, transitions, edges, and tokens. Places are represented graphically by circles, transitions are represented by rectangles, edges are represented by directed arrows, and tokens are represented by little solid (filled) circles. Petri nets may be extended in a variety of ways. These expansions include features such as the ability to describe probabilistic behavior, the ability to have weighted edges, and the ability to have tokens of multiple colors, among other things. Only the most fundamental Petri net ideas will be discussed.

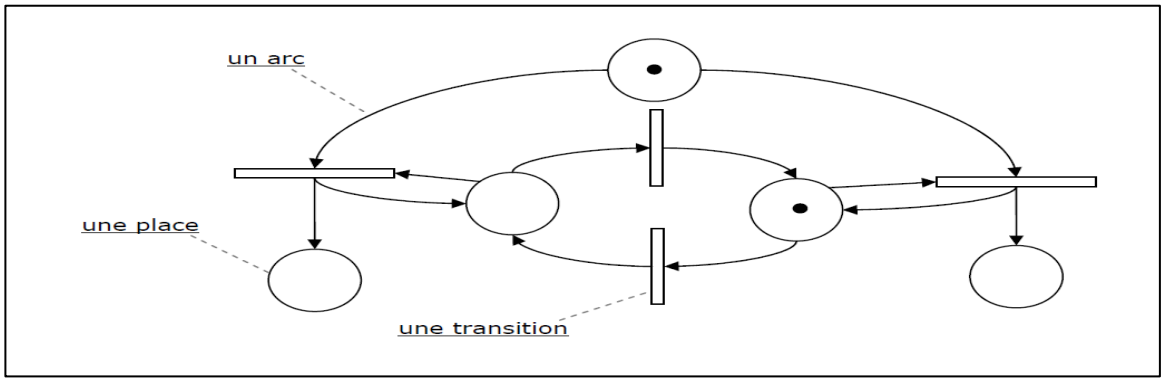


Figure 3.1: Simple Example of Petri Nets

3.3. Extensions of high-level Petri nets

Ordinary petri nets (figure 3.1) have a weakness in terms of expressive power. Indeed, we can see that the tokens in the spots don't contain any data and aren't distinguishable. As a result, tracking the route of tokens in a petri net like this is challenging. As a result, there are a number of Petri net extensions where we may solve this problem. We'll go through a few of these expansions in the next few sections.

3.3.1. Colored Petri Nets

Colored Petri nets [26], or colored nets, offer a more compact possibility of expression than ordinary Petri nets. In a colored network, a place contains typed tokens and a transition can be crossed in different ways. In the terminology of colored Petri nets, the types associated with places and transitions are called color domains and an element of a color domain is called a color. A marking of a colored Petri net associates to any place of the network a multiset of tokens typed by the color domain of the corresponding place.

An arc joining a place and a transition is labeled by a linear map called (color function). The color functions determine the typed tokens removed or added to a place for crossing a transition for a given color. That is to say, for the crossing of a transition instance (transition-color association).

Finally, the network transitions can be kept. A guard is a Boolean condition on the color of the transition that limits the crossing factor of the transition to the colors for which the guard is true.

The graphical representation of a colored network follows the same conventions as the graphical representation of an ordinary network, except that the labels of the arcs are no longer integers but functions of color. Transition guards are usually placed next to transitions and written in square brackets. The color domains of places and transitions will sometimes be omitted since they can generally be easily deduced from the color functions of the network.

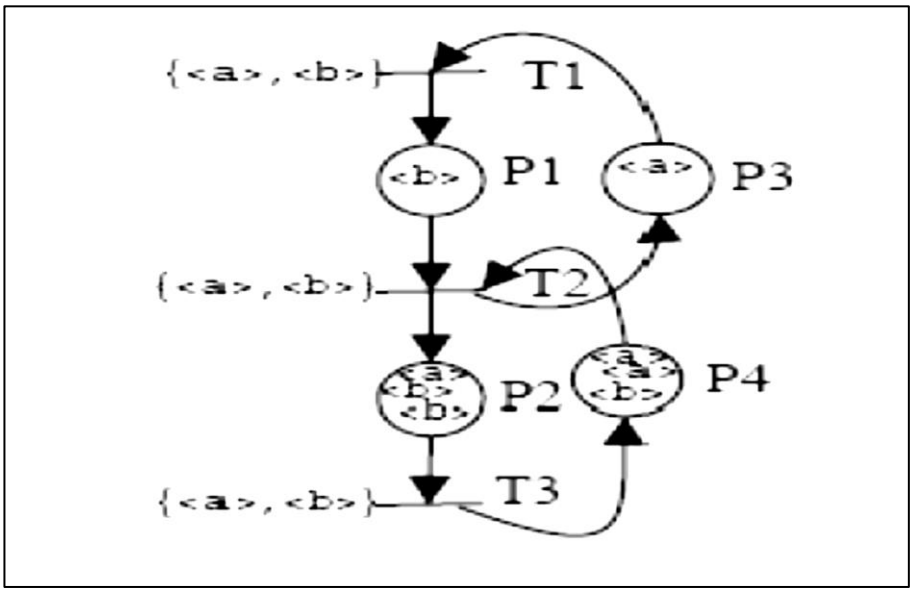


Figure 3.2: Colored Petri Nets

3.3.2. Time Petri Nets

Time Petri Nets (or Timed Petri Nets), introduced in [25] are one of the techniques proposed to specify and verify systems in which time appears as an important parameter [24].

Time nets are obtained from Petri nets by associating two min and max dates with each transition. Suppose t last became sensitized on date θ then t cannot be crossed before date **θ+min** and must be crossed no later than date **θ+max** unless crossing another transition has desensitized t before it is crossed. Crossing transitions is of zero duration. Temporal networks natively express specifications "in delays". By explaining the beginnings and ends of actions, they can also express specifications "in durations". Their field of application is therefore wide.

3.3.3. Timed Petri nets

There are many discrete event systems whose evolution depends on time. On the other hand, the notion of time is essential when one wants to evaluate the performances or to study the problems of scheduling of a dynamic system. The need to model and study such systems gave rise to timed Petri Nets [27].

In this Petri net model, the duration of an activity is explicitly integrated. The delay can concern the places (P-timed Petri nets) or the transitions (T-timed Petri nets) according to the modeled events.

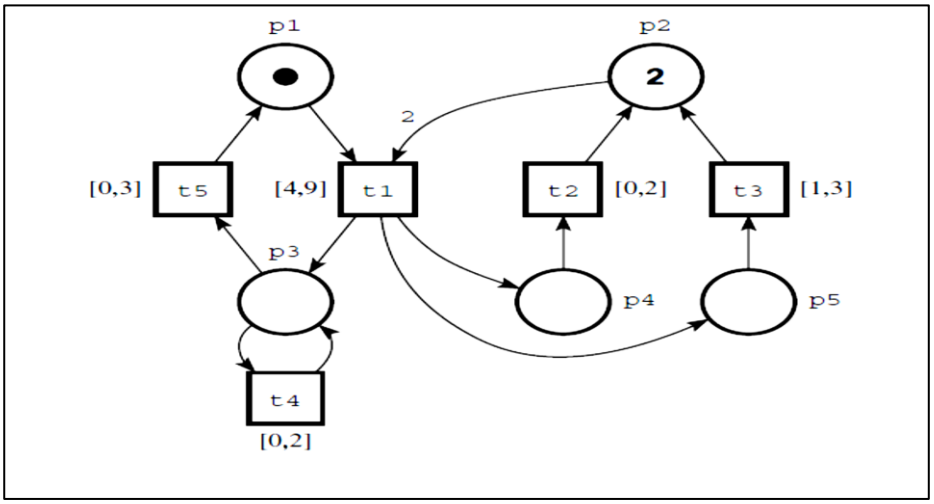


Figure 3.3: Time Petri Net

3.4. Paradigms for modeling mobile systems by Petri nets

As we said in the definition (recursive, adaptive, hierarchical, etc.), Petri Nets are a great tool for modeling a variety of systems. Mobile systems contain unique characteristics (migration, cloning, and so on) that are challenging to characterize with traditional Petri Nets. In what follows, we will describe three petri nets-based paradigms that we believe are excellent for modeling mobile systems.

3.4.1. The Nested Nets Paradigm

Nested Nets is a paradigm where tokens can also be a Petri net (same notion as the previous paradigm "Nets-within-nets") [29]. Nested Nets have value semantics based on an extension of colored Petri nets where tokens can change value without crossing transitions.

The Nested Nets paradigm [28] makes it possible to represent the concepts of mobility such as: locality, migration, cloning as well as the monitoring of the life cycle of a mobile agent. A Nested Net with two levels of abstraction and allows for example to model at level 0 the rentals, migration and cloning of mobile agents. Level 1 can be used to model the life cycle of a mobile agent.

3.4.2. The Nets-within-nets paradigm

This paradigm formalizes the fact that Petri net tokens can also be Petri nets. Using this perspective, it is possible to model hierarchical structures in a graceful manner.

Places and transitions are used to create a net. A net's active component is its transitions. Rectangles or squares are used to represent transitions. Arcs leading from locations to transitions and from transitions to places define this.

The introduction of this paradigm offers us a simple and easy way of modeling systems with recursive, hierarchical or mobile characters. A Nets-within-nets has two levels of abstraction:

- A higher level called system network where the tokens in these networks are themselves Petri Net.

- A lower level called object network, the network at this level is considered as a token in a system network, the places of an object network contain simple tokens.

The token nets are "lying" as tokens in locations exactly like conventional (black or colored) tokens, according to the logic of nets inside nets. Figures 3.4 and 3.5 show how this is done. When modeling larger networks, a display like the one shown in the illustrations is impractical. As a result, the modeling tool Renew implements a pointer concept: net tokens are pointers (hence the name) to individual nets, each of which is presented in its own window[32].

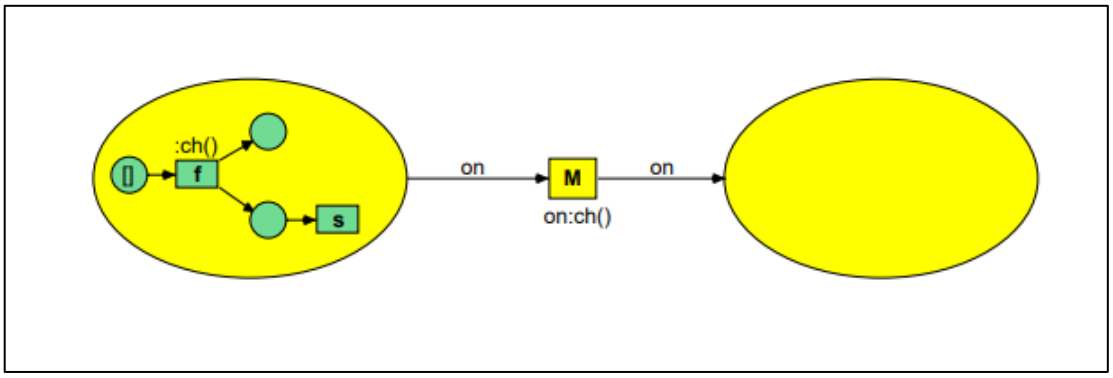


Figure 3.4: Object net embedded in system net

Consider a case in which there is a two-level hierarchy. The net token is thus referred to as "object net," while the surrounding net is referred to as "system net." Figure 3.4 depicts the situation: The arc inscription on can be attached to the object net in the left location of the system net. As a result of this putative binding, transition **M** is activated. A synchronous channel (**on:ch()**) is also engraved on M. This inscription signifies that in order for the object net on to become a real binding of transition **M**, a suitable counterpart must be located inside on a channel:ch enabled transition () [32].

This precondition is fulfilled by transition f of the object net. So, the synchronous firing of object and system net can take place and leads to the situation in Figure 3.5.

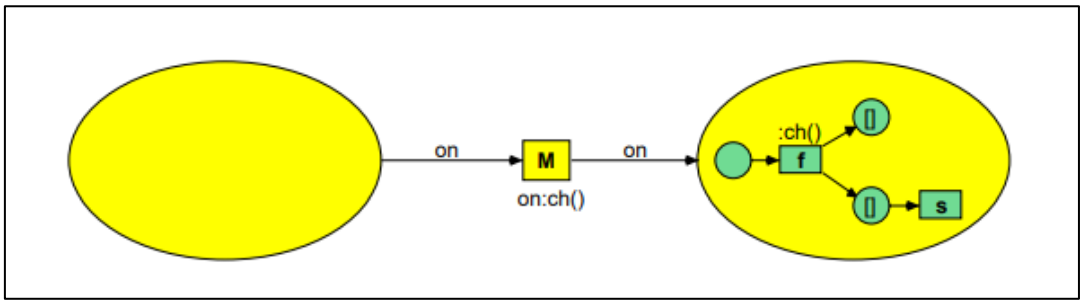


Figure 3.5: Object and system net after firing

The object network is relocated to the right of the system network. The marking of the object net changed synchronously, making further firing of transition f impossible. The example shows how the interaction between object and system nets can be used to model mobile entities moving through a system net, where the system net allows or disallows movement while the mobile object net moves at the appropriate time by activating a transition that is inscribed with the channel of the system net's move transition [32].

3.5. Types of Mobility

The interaction between the object net and the system net results in four different ways for an object net to move or be moved. The object net is moved within the system net, and neither the object nor the system net is in control of the movement (Spontaneous Move), and it's forced to move by the system net (Transportation, Objective Move) [32].

The movement is triggered by the object net, which has no impact on the system net (Subjective Move), and it's agreed upon by both networks (Agreed Upon Action). The interaction between object networks and system networks induces four possibilities to control the type of movement and therefore the mobility of object networks [32].:

- Weak mobility: Only data and state of an agent is transmitted, a new agent is invoked at the target platform.
- Strong mobility: The agent is transmitted together with its runtime environment (e.g., data, state, stack trace, and so on).

Strong mobility is difficult to implement using a programming language like Java [29]. Inside a Petri net simulator, it is the most natural form of migration. Enabling strong mobility between different simulators is a little bit more difficult, but no real problem at least this holds for the Petri net simulator Renew. Weak mobility as a special kind of strong mobility is possible anyway. Therefore, we can abstract from the implementation details of the migrations and focus on the higher-level interactions between mobile entity and environment.

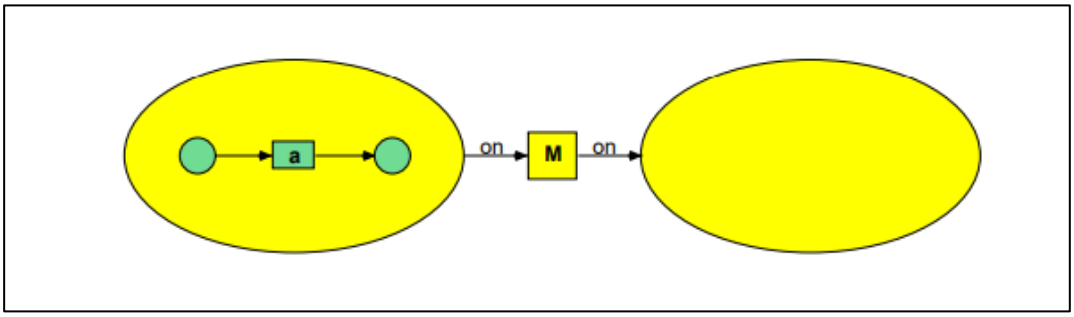


Figure 3.6: Spontaneous Move

Spontaneous Move [32]. It's possible that the movement will happen on its own if neither the object net nor the system net impacts it. Figure 3.6 depicts the problem. The movement has no pre-existing negative effects.

Subjective Move [32]. One may argue that the second possibility does not exist since item nets can migrate from one area to another using the system net. As a result, the system net "decides" which motions are allowed and which are not. However, in a particular system net, an object net can regulate mobility, as demonstrated in Figure 3.7.

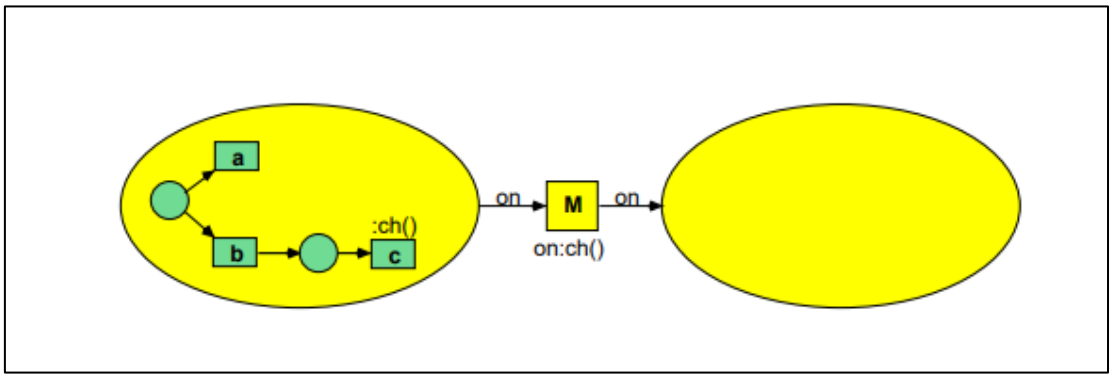


Figure 3.7: Subjective Move: Object net triggers movement

The synchronous channel is the only need for the movement to be carried out in the diagram (see transition M). If its counterpart inside the object net is likewise active, the synchronous channel is engaged (that means, it is inscribed to an enabled transition). As a result, mobility is only dependent on the (firings of the) object net [32].

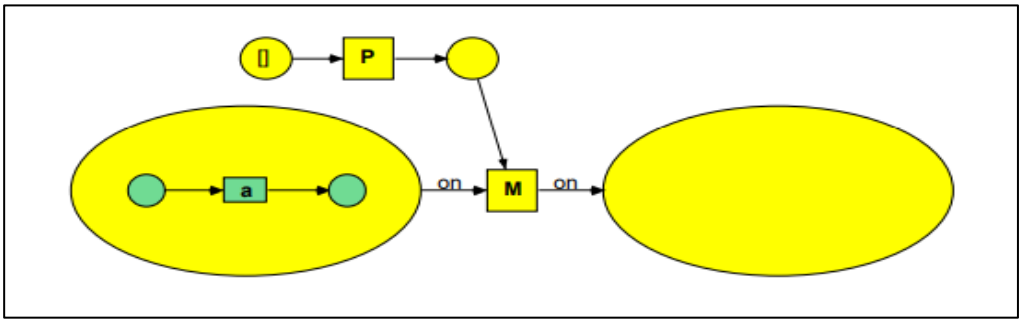


Figure 3.8: Transportation: System net triggers movement

Objective Move[32]. The system net may regulate the movement, and the object net is moved from one point to another if Figure 3.6 is expanded with some form of condition for transition **M**. Figure 3.8 depicts one possible method for the system net to regulate the transition **M**; another option is to use a guard (see Figure 3.9).

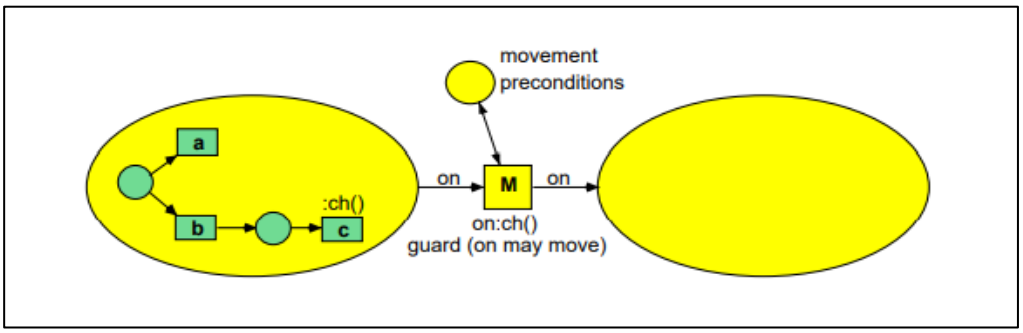


Figure 3.9: Consensual Move

Both the object and the system net impact transition **M** when Figures 3.7 and 3.8 are combined. They must agree on the movement in order for the transition to be possible. As a result, this type of movement is referred to as consensual. Figure 3.9 is an example of such a maneuver [32].

A combination of place holding movement conditions and an appropriate guard is shown in the figure as another approach to modeling a (side) condition for transition **M**. Transition **M** is only enabled if the object net is permitted to move, as the guard checks the movement circumstances [32].

3.6. Agent Systems

In the next part, we'll apply the broad principles of how mobility may be modified to a specific type of multi-agent system. The simulation of a mobile agent (object net) traveling across a "world" of several places (system net) allows for a natural recreation of real-life circumstances. First, the MULAN multi-agent architecture is shown, which benefits from the utilization of nets inside nets as well.

The MULAN multi-agent architecture [27] is based on the nets-within-nets paradigm. This architecture is used to describe the natural hierarchy of an agent system. The MULAN architecture facilitates the modeling of agents, especially if these agents are mobile. Figure 3.9 presents the general structure of the MULAN architecture. This figure is divided into four parts where each describes a level of abstraction in terms of a system network.

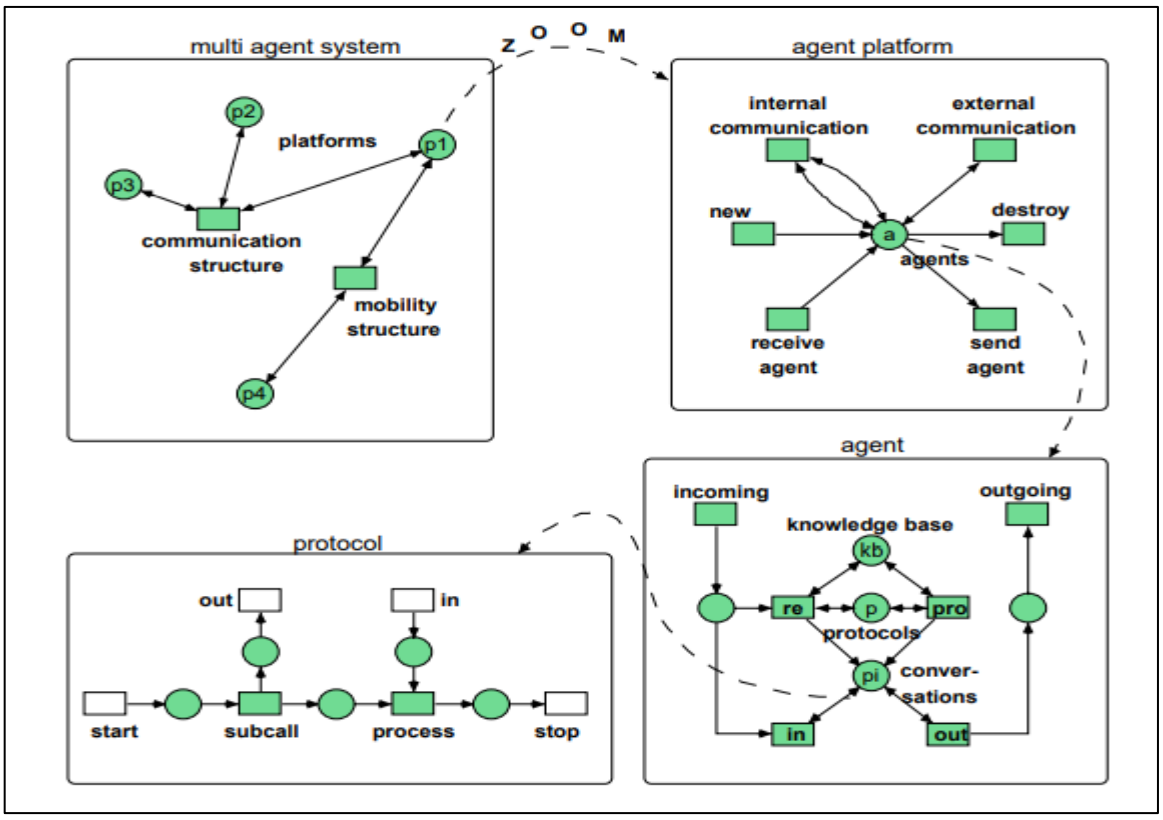


Figure 3.10: Agent systems as nets within nets

Part 1 of Figure 3.10 shows the organization of an agent system, with locations serving as tokens (each platform containing one or more agents) and transitions serving as communication or mobility routes between them. platforms for a system

We get the network in part 2 of figure 3.10 by zooming in on a platform from part 1. This section describes the internal structure of an agent system's platform. We can see from this structure that a platform provides a variety of services to the system's agents. The transition (new) allows a new agent to be created, whereas the transition (destroy) allows it to be destroyed. The two transitions (internal communication) and (external communication) enable for the exchange of messages between the system's actors in terms of communication. Internal communication (between agents on the same platform) and external communication (between agents on separate platforms) are both feasible

(between agents of different platforms). Agents can transfer from one platform to another using the two transitions (accept agent) and (send agent).

The structure of an agent is seen in part 3 of figure 3.10 It's worth noting that the agents are likewise portrayed as a network. The agent's knowledge base is stored in the location (knowledge base). It specifies the agent's intelligence level. The agents' actions are specified using protocols (also in the form of a network) that are used as models in the environment (protocols). The instantiation of a protocol template might be triggered by the receipt of a message or a change in the context around the agent. The knowledge base of the agent always influences the protocol to be instantiated. The instantiated protocol is saved in the specified location (conversation). Figure 3.10, part 4, is an example of a running protocol.

3.7. Conclusion

We tried to introduce the Petri Net paradigm and its extensions in this chapter, starting with a description of Petri Nets in a general, formal, and informal framework, then these extensions, and finally the modeling and implementation of our case study in the following chapter.

Chapter 4

Modeling and implementation

4. MODELING AND IMPLEMENTATION

4.1. Introduction

We present a model-based testing strategy for multi-agent systems in this chapter, based on a model called Reference Network, and we construct a tool to provide a standard and automated approach. A brief case study is used to show the feasibility and interest of the suggested technique. Researchers have been more interested in model-based testing, especially since models have become more prevalent in software design and development. The process usually consists of four steps:

- creating an abstract model of the system to be tested.
- validating the model (either using animation or formal proof).
- creating abstract tests from the model.
- running the tests on the system and forming a judgement and ultimately, the evaluation of the outcomes (oracle).

4.2. Proposed Approach

From the perspective of a tester, multi-agent systems can be divided into several levels of abstraction: the algorithmic level, the class, the agent, the society, and the system. For the first level of abstraction, traditional testing methodologies (functional, structural set of non-regression) have been fully implemented. Furthermore, there is a requirement for a framework to aid in the construction of automated testing. The scope of our proposed approach is depicted in Figure 18.

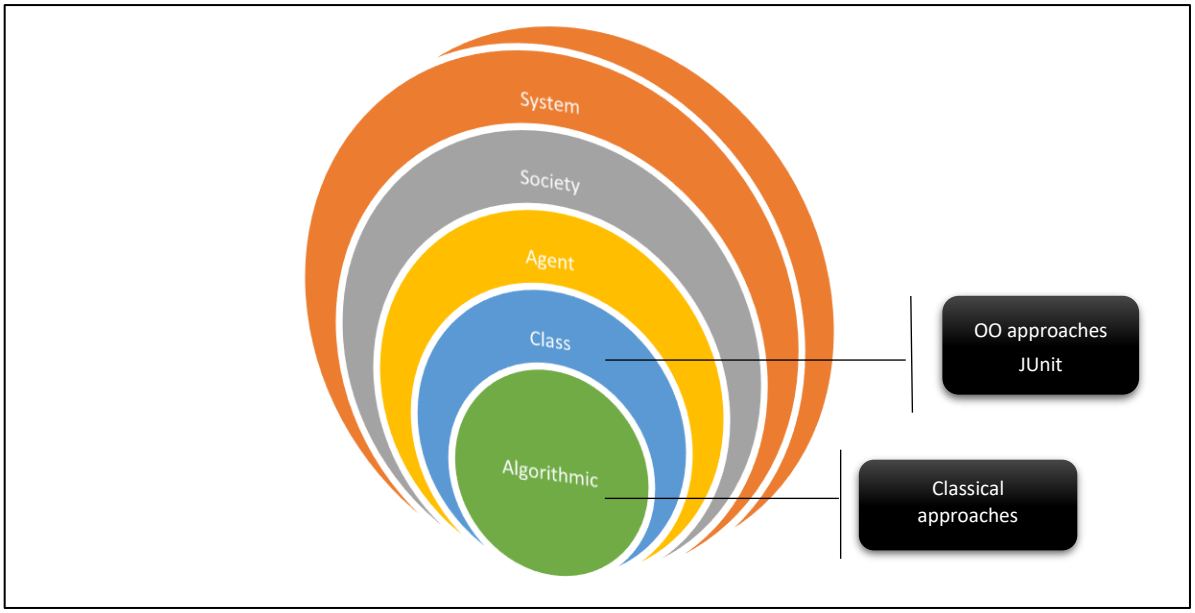


Figure 4.1: multi-level approach

The technique presented in this thesis is based on the model-junit tool, however it is tailored to multi-agent applications. Furthermore, although model-junit employs a finite state machine as a model, we use reference networks, which are a graphical representation that enables a flexible modeling method in which the tokens may be java objects and the networks can be considered objects.

Finally, the tester in model-junit must develop test cases that will be performed with the junit tool. The test cases in our technique are produced automatically from the model and are dynamically executable when the system under test is being executed, Figure (19) presents the general architecture of the proposed approach. It contains a series of four steps.

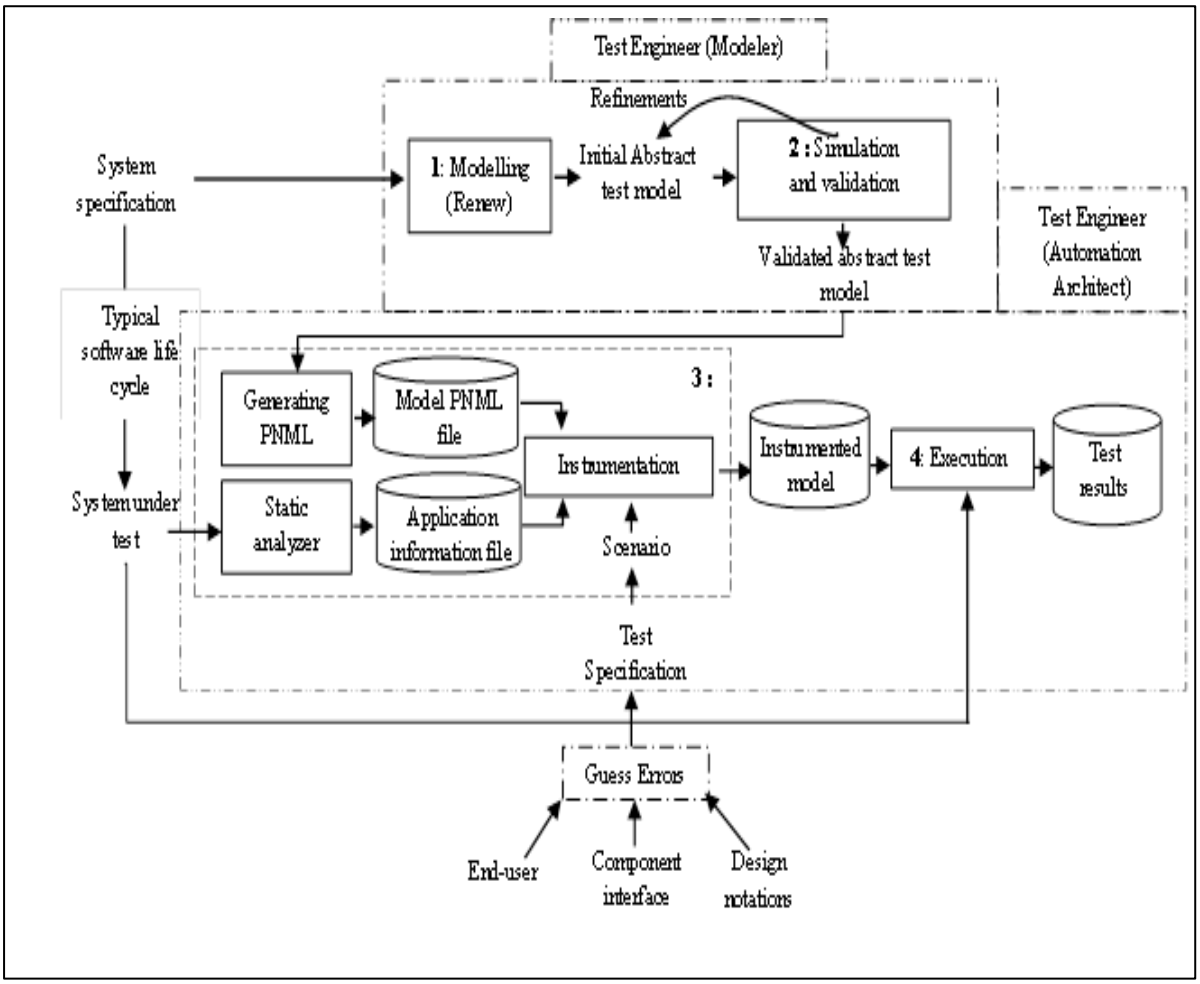


Figure 4.2: the general architecture of the proposed approach

We use the mobile agent model as a case study to better comprehend the suggested strategy and explain the many processes.

The modeling phase, as represented in rectangle (1), is the first stage in the model-based testing approach. We describe the abstract specification as a model at this step. The model is created to depict the system under test's expected behavior.

Phase (2), on the other hand, is required to validate the model's validity. The tester (modeler) must simulate his model for every potential situation, make corrections where necessary, and enhance the abstract test model until he is happy that it fits the basic criteria. During the test case execution step, errors that were not noticed during the validation phase might be detected.

Phase (3) represents the generation of test cases and their implementation. This is the most delicate stage, and it is the centerpiece of our offering. It cannot be carried out without the tested system's source code and is carried out under the tester's supervision. It takes as input one information file collected from the system under test and another from the test model: all the methods available in the source code and all the model transitions extracted in XML format (RefNet PNML format) and, of course, the scenario that the tester wishes to examine. A scenario like this is created based on the test requirements and the intended test levels (agent, company or system). Using a test case design approach known as "error-guessing," It depends on the tester's ability to guess where faults will be found in the system under test based on previous experience, knowledge, and intuition. test. The instrumentation's output is subsequently used to generate a model's news.

The execution phase is the final stage (4). During this stage, the tester might ask for the instrumented version of the model to be run. The predicted outcomes will be compared to the results achieved after executing a test sequence.

4.3. Case study

To better understand the proposed approach and explain the different steps, we present the mobile agent model as a case study.

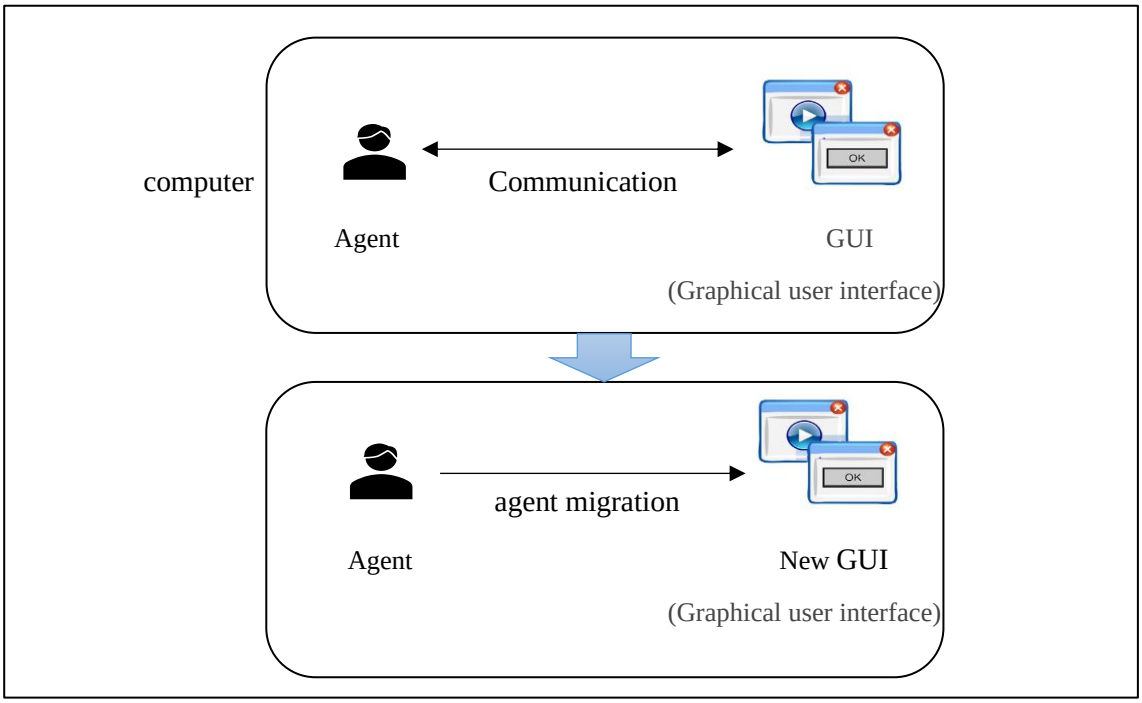


Figure 4.3: Mobile agent case study

In this example of mobile agent and because this agent has a GUI, the communication between the agent and its GUI based on events passing.

The agent can move only if there is an available location which requested from the ASM, those available locations will appear in the GUI as a list, this list might be stored in the agent and migrates with it. A new GUI will be created and sets the list of visited locations and the list of available locations (via the behaviour) in the GUI as soon as the agent arrives to the new destination.

The suggested approach's step sequences are as follows:

4.3.1. Modeling

The production of a model dedicated to the test is the main activity of a user of the model-based test technique. This involves designing a model from which the test cases to be executed on the system under test will be produced. This model represents the behaviors of the system at a certain level of abstraction. The goal of the test model is twofold:

On the one hand, it specifies the expected behaviors of the system under test and thus simplifies the test verdict phase. This phase consists of comparing the results obtained following the execution of the system with the expected results. On the other hand, its structure is exploited during the generation of test cases to guarantee a certain coverage of the model and therefore of the functional specification from which it comes. Typically, the different families of structural coverage criteria can be used to guarantee a given model coverage.

Figure 4.4 shows the abstract models of each level of the system under test. The system level is represented by (a), (b) represents the agent level and (c) the protocol level.

In the “system level” we have two methods <<clone()>>and << move()>>, with <<clone()>> method the agent cloning itself to one of another available locations. The agent can migrate to another GUI with <<move()>> method.

Before those methods work it must have been a communication between the agent “agent level” and the GUI “protocol level”.

With such a paradigm, the modeling phase is easier. Indeed, during the modeling of complex systems, it is strongly discouraged to want to manage all the complexity of the system at the time of its modeling and/or its execution. Several views are possible: the multi-agent system as a whole, the platforms on which the agents are positioned, the agent itself or simply its behavior.

4.3.2. Validate the model

Since this abstract formal test model is manually derived from the system requirements specification, validation of the test model against the system requirements specification should be done first in order to find major errors in the test model test.

With the MBT approach, if some errors persist in the model, they are very likely to be detected when the generated test is run on the system under test. Renew relies entirely on simulation to explore the properties of a network, where the tester can dynamically and interactively explore the state of the simulation. Indeed, 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 network, the tester has the opportunity to gradually correct and refine the abstract test model. Such a robust and easy to use tool greatly reduces the large initial effort in terms of man-hours required primarily in building and validating the test model.

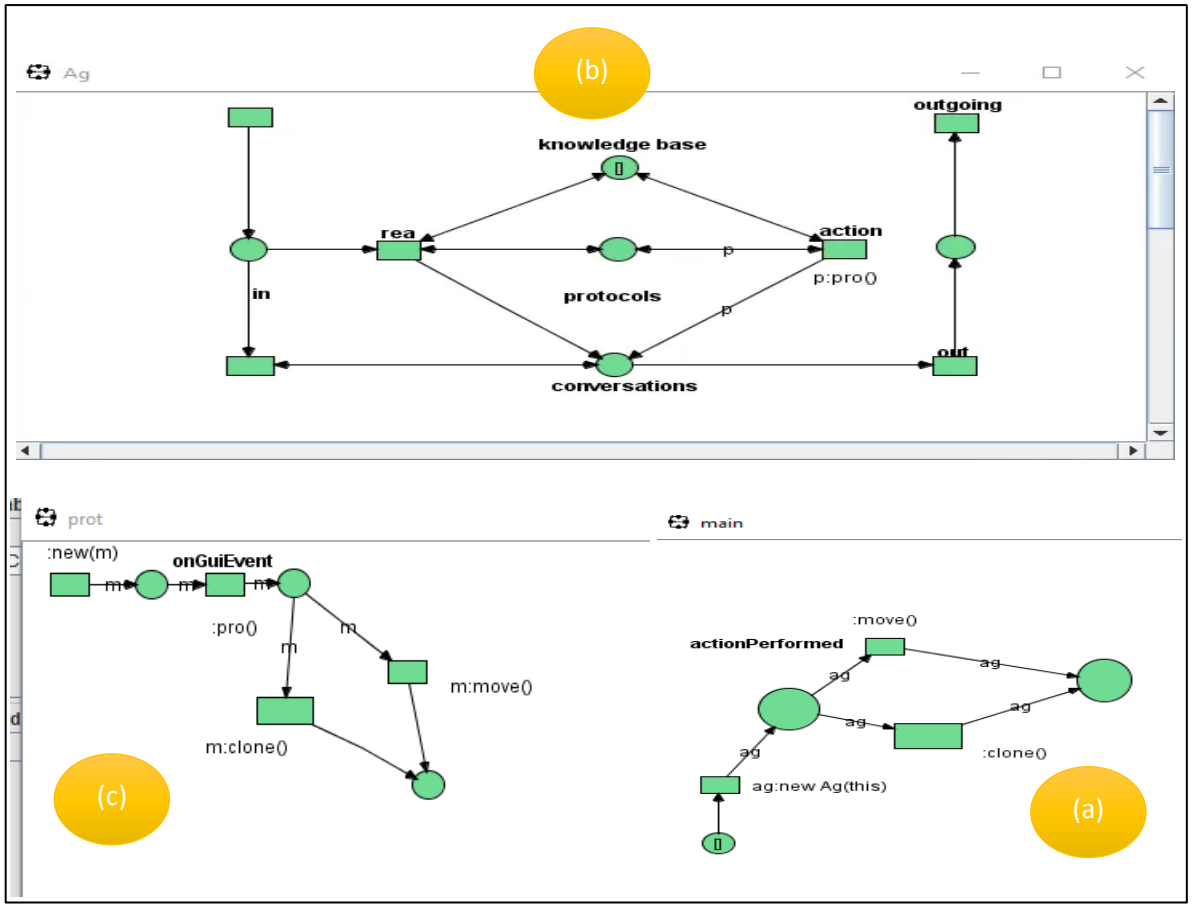


Figure 4.4: Modeling of the mobile agent system.

4.3.3. Generation and realization of test cases

In the literature [31], test case generation approaches have suffered from the problem of combinatorial explosion of the number of test cases. For economic reasons, it should be kept to a minimum. For quality reasons, it must be high enough to reduce the number of failures remaining in the field to an acceptable number.

To cope with this explosion, we opted for the well-known test case generation techniques. Our approach suggests that testers use the test case design technique called "error guessing". This technique is based on the tester's ability to draw on experience, knowledge and intuition to locate faults. The test engineer can thus enumerate a list of situations with a high risk of error and prepare one or more execution scenarios to verify them.

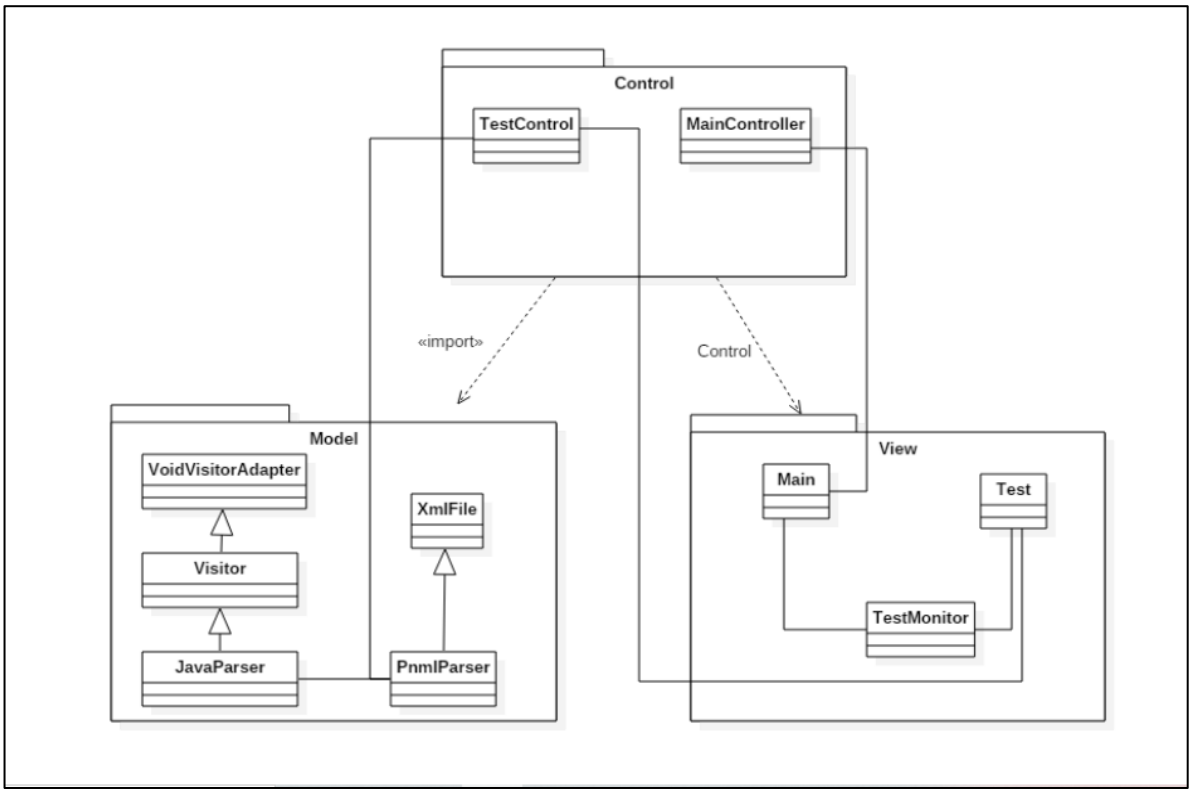


Figure 4.5: Class diagram of the test monitor [34]

In addition, the concretization phase has the main task of transforming the abstract test cases into concrete test cases. Subsequently, these concrete test cases can be submitted to the application under test.

The objective is to reduce the gap between the abstract model and the concrete system by adding missing information and transforming the abstract entities into test platform language structures. But before realizing the generated test case, some preparatory work must be done.

This is static analysis it concerns both the source code of the application to be tested and the networks of the test model. Concerning the system under test, the static analysis of the source code makes it possible to recover all the details (names of agents, names of classes, methods, parameters, types, etc.) in an information file of the application under test. Regarding the validated test model, all networks are translated into PNML files (RENEW's special format based on XML to represent the models).

The result of the static analysis is shown in Figure 23. In our test case the coverage of place(s), transitions and arc(s) are calculated at runtime on the one hand and on the other hand also the number of methods, class and agent are calculated

4.4. Technical choices

The best way to build a multi-agent system is to use a multi-agent platform. A multi-agent platform is a set of tools needed to build and provision agents within a specific environment. There are currently several platforms for the development of multi-agent systems.

We chose the Java programming language and the SMA JADE development platform for the following reasons: It is easy to create agents with jade. Jade handles communication between agents and provides agent management interfaces. The direct communication mode (with messages) is supported by the JADE platform using the FIPA ACL language. The Jade platform caters to multiple features and offers a wide range of libraries. The agents developed in JADE are written entirely in JAVA which is an easy language based on the notion of object. JAVA was chosen for its rich tools facilitating its use.

The programming was carried out under the Eclipse development environment, which in recent years has reached maturity and offers exemplary richness and ergonomics. And we have the source code of our application under test (SUT).

We used the test monitor MATT (Multi Agent Testing Tool)[34] for do the static analysis of the source code which makes it possible to recover all the details (names of agents, names of classes, methods, parameters, types, etc.) in an information file of the application under test. As shown in Figure 4.6.

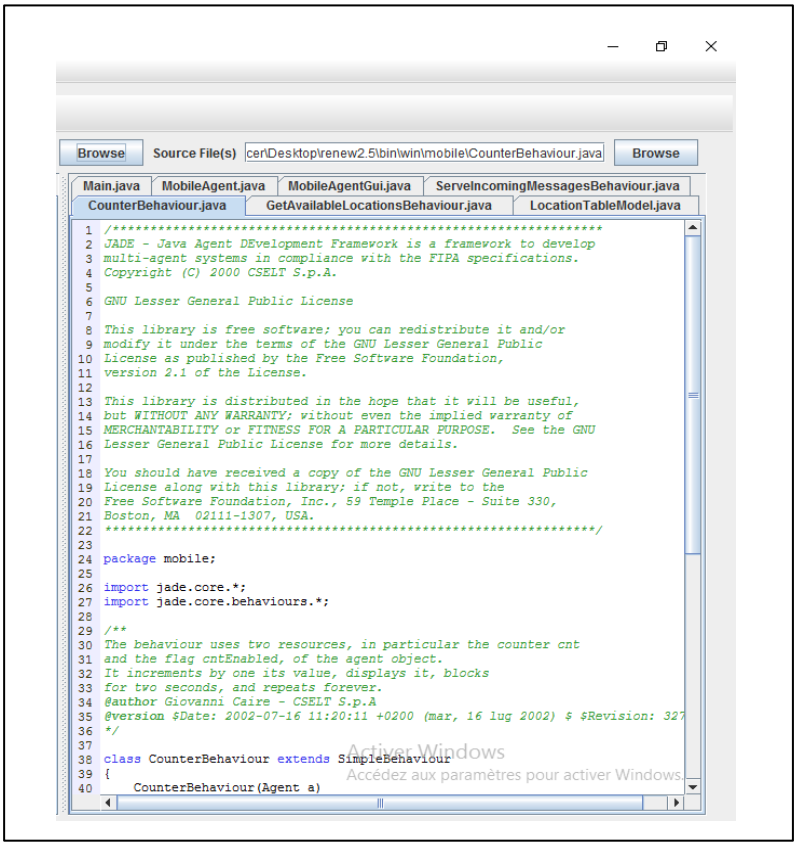


Figure 4.6: Result of the static analysis of the source code.

For the validated test model, all networks are translated into PNML files (Renew's special XML-based format for representing models). See Figure 4.7.

Once all the inputs (system information file, model information file and test case) are available, the test monitor proceeds to instrument the test case by adding in the model a certain set of determined routines.

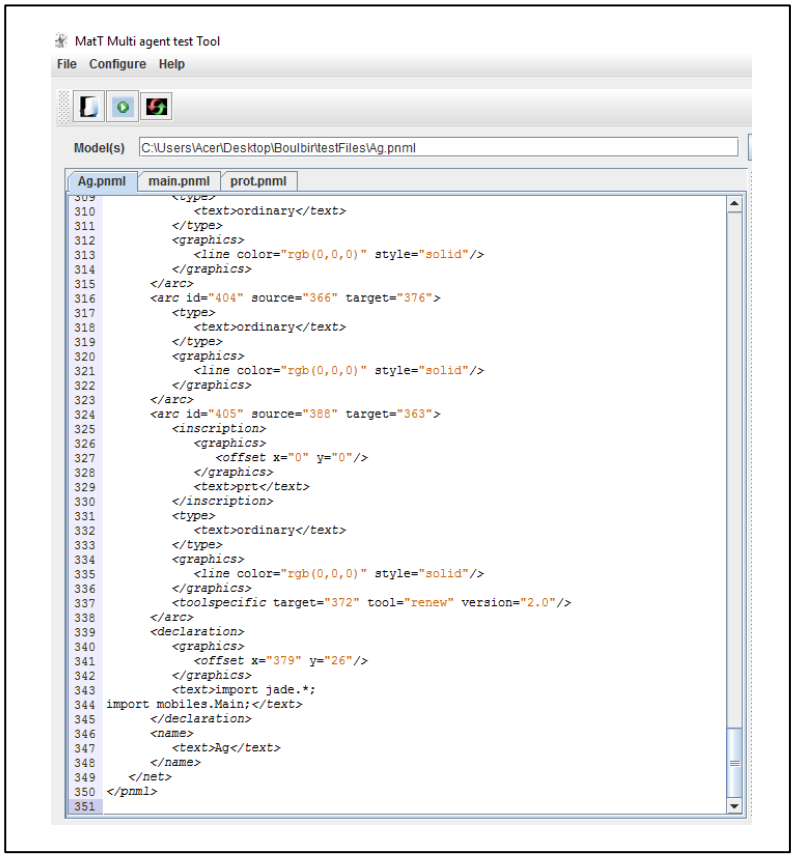


Figure 4.7: Result of the static analysis of the model.

More precisely, the tester selects one of his transitions from the model with its corresponding function in the source code. Then the test monitor adds the data (automatically) into the model for that transition. Figure 4.8 shows the instrumentation module.

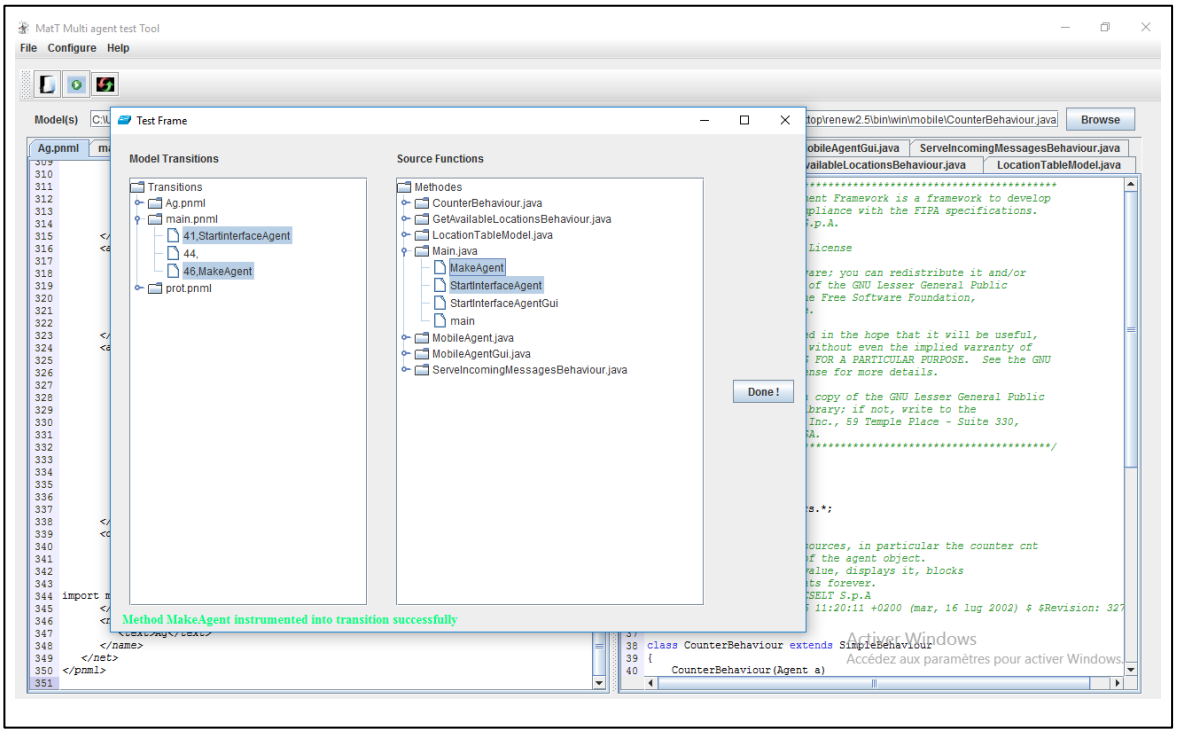


Figure 4.8: Instrumentation of the model

Figure 4.9 shows the instrumented version of the model where all the adding instruments are circled. The reader can easily compare the original figures of the model to Figure 4.9. Finally, because this operation (instrumentation) is derived automatically, it greatly reduces the testing effort.

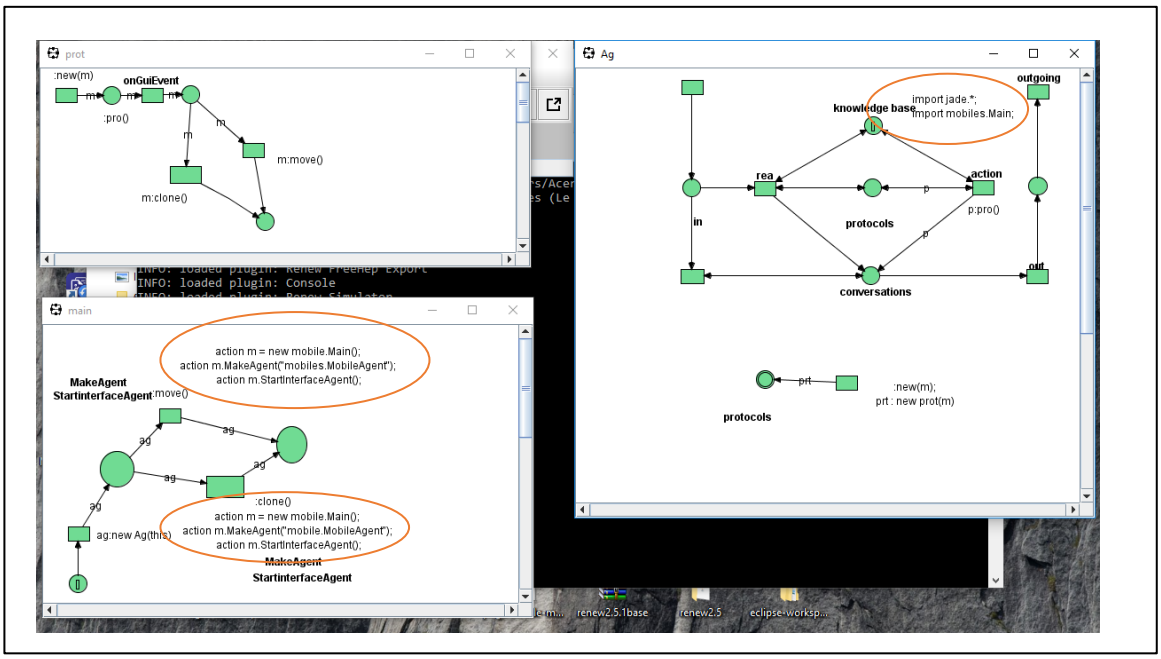


Figure 4.9: Instrumented model version

4.5. Calculating Coverage

By using PNML files and java files, we can calculate the coverages as shown in Figure 4.8 by extracting the names of each method, transition, place and arc.

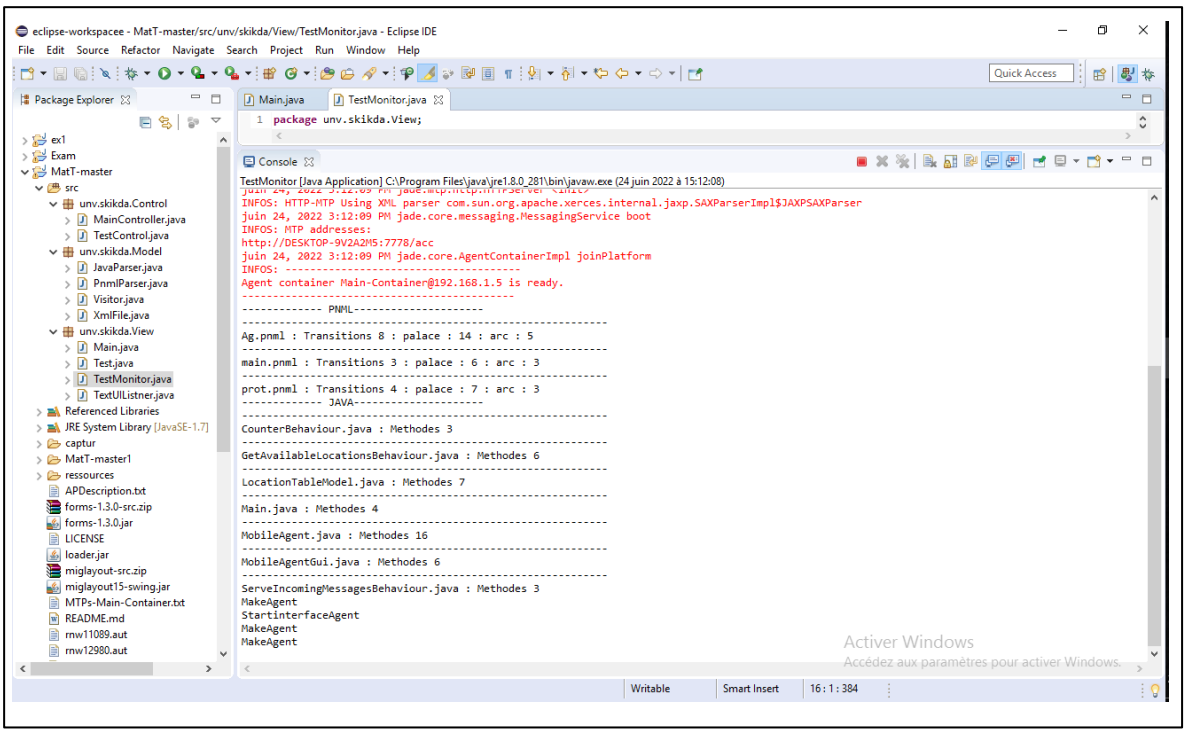


Figure 4.10: the coverages

4.6. Execution and evaluation

Now the instrumented version of the model under test is ready to be executed. The results obtained following the execution of a test sequence will be compared with the expected results and a verdict is formed. Figure 4.9 shows the execution of the mobile agent example.

An animation (execution) of the instrumented model calls the functions concerned by the test in the application. If the test passes correctly the model continues the animation to a new place or transition. (a) represent the execution result of the method <<move()>>, (b) represent the execution result of the method <<clone()>>.

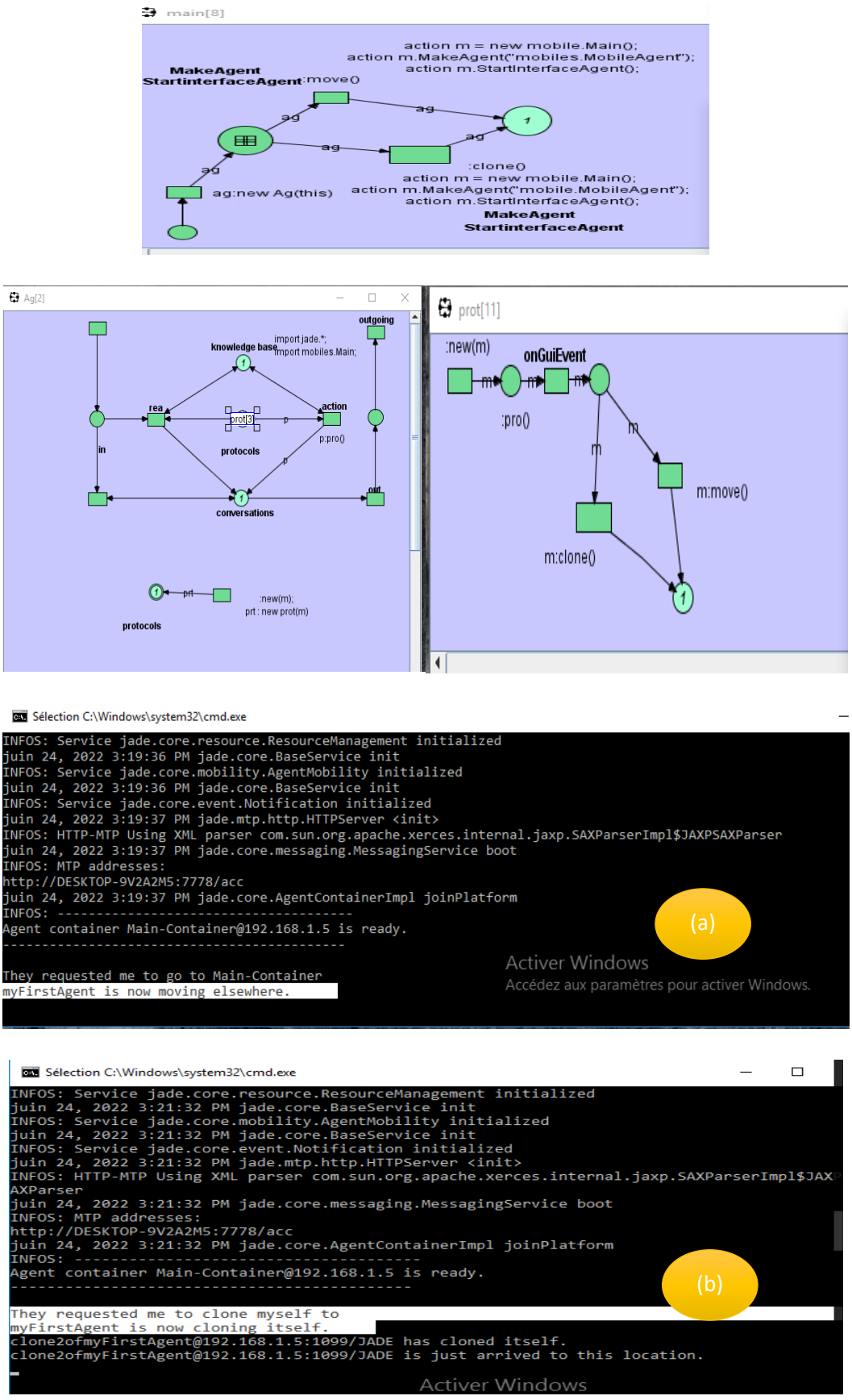


Figure 4.11: Execution and evaluation of results

4.7. Discussion

[33] have presented a comprehensive review of state of art on test suite reduction frameworks. In their survey, taxonomy, for the thematic classification of test suite reduction frameworks and tools, uses 10 parameters is proposed (see [33] for detailed discussion of these parameters).

Although that the proposed approach cannot be considered as full test suite reduction technique, but we think that using those parameters as a classification method will permit to situate the proposed approach against other frameworks and approaches. Tables 1 and 2 show the values of the 10 parameters when considering our approach. They are highlighted with gray color to distinguish them from other possible values.

Furthermore, the study classifies existing frameworks and tools into 5 main classes based on the targeted testing domains and approaches:

(i) randomized unit testing.

(ii) user session testing.

(iii) retargeted compilers testing.

(iv) integer linear programming.

(v) automated fault-localization.

It is worth noting that our approach can be considered as special kind of randomized unit testing because of use of some random elements in selecting test cases.

	Approach	**Testing Paradigm**	**Optimization type**	**Coverage source**	**Execution platform**
The proposed approach	Coverage Based	Structured	N/A	Source Code	N/A
	Search Based	Object Oriented		Test Execution	
	ILP	Aspect Oriented		Model	
	Similarity Based	Multi Agent Oriented			

Table 4.1: Parameters values for the proposed approach

	Computational mode	License	Evaluation	Customizable	Support
The proposed approach	N/A	Commercial	External	Full	Doc + Executable
		Research		Partial	Doc + website
		Free	Internal	No	Doc + Executable + Website

Table 4.2: Parameters values for the proposed approach (continued)

To conclude, our running example has focused on a simple form of agent cooperation which is a stop-and-wait protocol. That is the producer transmits a data packet at a time and must wait for an acknowledgement from the receiver. What happen if the transmitted packet is lost? In such a situation, the sender must retransmit packets and keep track of the data packet currently being sent. Alternatively, the receiver should keep track of the data packet expected next.

To tackle such extended transmission protocol, developers have to introduce changes in the actual system under test, which will create a new product variant. In traditional testing techniques, even small changes to the application functionality can render generated test cases useless and need to be completely re-generated to capture the new changes in the system under test functionality.

In the proposed approach when system functionality changes or evolves, all we need to change is the abstract test model. After that, the automation architect has only to design scenarios that precisely invokes the modified parts of the system under test. No need to retest the unchanged parts of the system under test.

Experience shows that failures that occur when the tests are run are likely to be due to errors in the model or errors in the implementation. So, the process of model-based testing provides useful feedback and error detection for the requirements and the model as well as the system under test.

Different coverage criteria (place coverage, arc coverage and path coverage) can be measured and calculated which determines which tests are interesting or not.

4.8. Conclusion

In this chapter, we used the mobile agent as a case study to demonstrate a model-based testing approach for a multi-agent system using the reference network paradigm.

The suggested approach is strongly reliant on the Renew tool's simulation findings. As a result, we used a test monitor (MATT: Multi Agent Testing Tool)[34] that takes the test model and the system to

be tested as input and runs the full test procedure. A static examination of the internal structures of the agents to be tested, more precisely, provides MATT with a global picture of how the agents are built. After that, the data is entered into the test model. The model's execution (at the same time as the program under test) allows you to see if the system's answers match the predicted outcomes.

General conclusion

In current culture, where software is more prevalent in our everyday lives, software quality is becoming increasingly important. To characterize and discover faults in software, several verification approaches have been created. Testing is the most often used approach.

The goal of software engineering is to save money, decrease development time, and assure a high quality of releasable software. This is a dream for everyone, whether they are customers, users, developers, or testers, because the goal of testing is to assess quality rather than to enhance it. Its goal is to evaluate or execute a program in order to find as many flaws as possible and expose any potential issues. or the conduct isn't what you'd anticipate. Testing is still an effective way to assess software or a software component's trustworthiness.

The reference network paradigm was utilized to propose a model-based testing approach for a multi-agent system, with the producer and consumer as a case study. The modeling process concludes not only with a functioning system model, but it also supports all stages of the model testing approach, which is a strong argument in favor of utilizing the networks paradigm in networks. This considerably lowers the primary drawbacks of model-based testing methodologies while also assisting testers in creating and running tests in a consistent and automated manner.

The powerful and simple-to-use Renew tool considerably minimizes the large initial work required to create and validate the test model in terms of man-hours. Furthermore, the Mulan architecture's four-layer structure (multi-agents, platform, agent, and behavioral protocols) of the system.

Referances

[1] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries; IEEE; New York, NY.; 1990.

[2] J. Nielsen (1994). Usability Engineering, Academic Press Inc, p 165

[3] R. Savenkov (2008). How to Become a Software Tester. Roman Savenkov Consulting. p. 168. ISBN 978-0-615-23372-7.

[4] Code Coverage Analysis (bullseye.com)

[5] M. Blackburn, R. Busser, A. Nauman. Why Model-Based Test Automation is Different and What You Should Know to Get Started. Software Productivity Consortium, 2004.

[6] I. K. El-Far. Enjoying the Perks of Model-Based Testing. STARWEST, 2001. Available: http://www.geocities.com/model_based_testing/perks_paper.pdf.

[7] M. Utting and B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, 2007.

[8] A. Petrenko: Fault Model Driven Test Derivation from Finite State Models: Annotated Bibliography; in: Proc. of the 4th Summer School on Modeling and Verification of Parallel Processes (MOVEP2000), Springer LNCS 2067, 2000.

[9] A. Pretschner, W. Prenning, S. Wagner, C. Kühnle, M. Baumgartner, B. Sostawa, R. Zälch, T. Stauner. One evaluation of Model-Based Testing and its Automation. ISCE'05, ACM, 2005.

[10] E. Farchi, A. Hartman, S. S. Pinter. Using a Model-Based Test Generator to Test for Standard Conformance. IBM Systems Journal, 2002.

[11] E. Bernard, B. Legeard, X. Luck, F. Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. 2004.

[12] I. K. El-Far, J. A. Whittaker. Model-Based Software Testing. Encyclopedia on Software Engineering, Wiley, 2001.Available:

[13] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton. ModelBased Testing in Practice. Proceedings of ICSE'99 (ACM Press), 1999. Available:

[14] Caire, G., Cossentino, M., Negri, A., Poggi, A., Turci, P., Multiagent systems implementation and testing. In Proc. Of 4th International Symposium - From Agent Theory to Agent Implementation (AT2AI-4), 2004.

[15] Filman, R., Elrad, T., Clarke, S., Aksit, M. Aspect-Oriented Software Development. Addison-Wesley, 2005.

[16] Kiczales, G.,Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J. Aspect-Oriented Programming. European Conference on Object-Oriented Programming (ECOOP), Springer, LNCS (1241), 1997.

[17] Garcia, A., Lucena, C., Cowan D. Agents in Object-Oriented Software Engineering. Software Practice & Experience, Elsevier, 34 (5), pp. 489-521, 2004.

[18] Binder, R. Testing object-oriented systems: models, patterns, and tools.Addison-Wesley Longman Publishing Co., Inc., 1999

[19] Mackinnon, T., Freeman, S., and Craig, P. EndoTesting: Unit Testing with Mock Objects. Proc. of XP2000, 2000.

[20] Myers, G. J. The Art of Software Testing. Wiley, 2nd Ed. 2004

[21] Marie-Pierre Gleizes, Jorge J. Gomez-Sanz (Eds.) "Agent-Oriented Software Engineering X" 10th International Workshop, AOSE 2009 Budapest, Hungary, May 11-12, 2009 Revised Selected Papers

[22] Moreno, M., Pav´on, J., Rosete, A.: Testing in agent oriented methodologies. In: Omatu, S., Rocha, M.P., Bravo, J., Fern´andez, F., Corchado, E., Bustillo, A., Corchado, J.M. (eds.) IWANN 2009. LNCS, vol. 5518, pp. 138–145. Springer, Heidelberg (2009)

[23] Nguyen, C.D.: Testing Techniques for Software Agents, PhD thesis, International Doctorate School in Information and Communication Technologies - University of Trento (2008)

[24] Réseaux de Petri temporels : méthodes d'analyse et vérification avec TINA http://projects.laas.fr/tina/papers/etr06.pdf.

[25] MERLIN P. M., *A Study of the Recoverability of Computing Systems.*, Irvine: Univ. California, Thèse de doctorat, 1974.

[26] Christophe Pajault, Model checking parallèle et réparti de réseaux de Petri colorés de haut-niveau

[27] Wael Khansa, Réseaux de Petri P-Temporels contribution à l'étude des systèmes à évènement discrets.

[28] Mohamed Rédha BAHRI, Une approche intégrée Mobile-UML/Réseaux de Petri pour l'Analyse des systèmes distribués à base d'agents mobiles

[29] Kees V.H, Irina A. Lomazova,Olivia Oanea, Alexander Serebrenik, Natalia Sidorova, Marc Voorhoeve, Nested nets for adaptive systems

[30] Lorenzo Bettini and Rocco De Nicola. Translating strong mobility into weak mobility. In Gian Pietro Picco (Ed.), Mobile Agents. 5th International Conference, MA 2001, Atlanta. Proceedings, volume 2240 of Lecture Notes in Computer Science, page 182 pp., Springer Verlag, Berlin, 2001

[31] A. Paul, and O. Jeff, "Introduction to software testing," Cambridge University Press, New York, NY, USA, 2008.

[32] Michael K¨ohler, Daniel Moldt, and Heiko R¨olke, Modelling Mobility and Mobile Agents Using Nets within Nets

[33] Khan S. U. R., Lee S. P., Ahmad R. W., Akhunzada A. and Chang V. (2016). A survey on Test Suite Reduction frameworks and tools. International Journal of Information Management, 36(6), 963-975

[34] S. Kerraoui, "Test des Systèmes Multi Agents dans les Environnements d'Intelligence Ambiante ", doctoral thesis, 20 Août 1955- Skikda University, 2019