electronics

MDPI

Article

A Novel Mutation Analysis-Based Approach for Testing Parallel Behavioural Scenarios in Multi-Agent Systems

Nour El Houda Dehimi *, Abdelhamid Haithem Benkhalef and Zakaria Tolba

Department of Mathematics and Computer Science, University of Oum El Bouaghi, Oum El Bouaghi 04000, Algeria
* Correspondence: dehimi.nourelhouda@univ-oeb.dz

Abstract: In this work, we propose a new test case generation approach that can cover behavioural scenarios individually in a multi-agent system. The purpose is to identify, in the case of the detection of an error, the scenario that caused the detected error, among the scenarios running in parallel. For this, the approach used, in the first stage, the technique of mutation analysis and parallel genetic algorithms to identify the situations in which the agents perform the interactions, presented in the sequence diagram, of the scenario under test only; these situations will be considered as inputs of the test case. In the second stage, the approach used the activities presented in the activity diagram to identify the outputs of the test case expected for its inputs. Subsequently, the generated test cases will be used for the detection of possible errors. The proposed approach is supported by a formal framework in order to automate its phases, and it is applied to a concrete case study to illustrate and demonstrate its usefulness.

Keywords: multi-agent systems; system level testing; mutation analysis; test scenario; parallel genetic algorithms; sequence diagram; activity diagram

Citation: Dehimi, N.E.H.; Benkhalef, A.H.; Tolba, Z. A Novel Mutation Analysis-Based Approach for Testing Parallel Behavioural Scenarios in Multi-Agent Systems. *Electronics* **2022**, *11*, 3642. https://doi.org/10.3390/electronics11223642

Academic Editors: Ricardo Santos and Byoungju Choi

Received: 29 September 2022
Accepted: 7 November 2022
Published: 8 November 2022

1. Introduction

The development of a system follows several stages, including requirement definition, design, implementation, testing, and deployment. In order to guarantee the quality of our developed system, we must have an effective testing strategy. Testing can be highly useful in identifying failures and validating the system being tested [1,2]. Unlike other types of systems, testing multi-agent systems (MAS) is a complex task due to the distinctive characteristics of the agents, such as responsiveness, proactivity, autonomy, and social ability [3]. Consequently, despite the rapid evolution of MAS, only a few proposals addressing MAS testing exist in the literature. Moreover, most of these propositions are related to unit-level tests [4–9] and agent-level tests [10–14]. Unit-level testing involves testing all the units that compose an agent, including code blocks, and implementing agent units such as goals, plans, knowledge base, reasoning engine, rule specification, and so on, in addition to ensuring that they work as expected. For the agent-level test, it consists of, on the one hand, testing the integration of the different units, tested at the unit level, inside an agent; on the other hand, it consists of testing the possibility of the agent achieving its goal within its environment. These two levels of testing, unit and agent, can ensure that agents operate correctly if they are run separately, but they cannot detect faults that might be created if agents are pooled.

The system-level testing ensures that, for its part, all agents in the system operate according to specifications and interact correctly. This level of testing, until now, aroused little interest from researchers compared to the interest shown in the first testing levels [15]. In fact, in recent years, only a few approaches have been proposed in the literature for system-level testing [16–20]. At this level of testing, the interactions between agents of the system under test can be performed in parallel, which causes, in the latter, the execution

of several behavioural scenarios, also, in a parallel direction. The possibility of executing several behavioural scenarios in parallel, by the agents of the system under test, complicates the test phase. Indeed, in the event of error detection during the test of a given scenario, it would be difficult or even impossible to locate and recognise the scenario that caused the detected error among the scenarios executed in a parallel manner.

To overcome this problem, it is mandatory to test each scenario individually to ensure that each error detected in a given scenario is not associated with another scenario operating in parallel. Therefore, it is necessary to generate test cases, where the inputs ensure the execution of the scenario under test only, namely, all the set of its interactions and all the set of its activities. These inputs should not perform any interactions or activities involved in the other scenarios. This represents the focal point of our approach.

The problem related to the possibility of executing several interactions and behavioural scenarios in parallel by the agents of the system under test was mentioned in the work [18]. The proposed approach, in this work, consists of using the constraints expressed in Object Constraint Language (OCL), necessary for the execution of each interaction to introduce plugs that can minimise the interval of inputs, so that one ensures the retrieved inputs cover only the desired interaction or the desired scenario. This work has brought promising results, but the use of OCL constraints necessary for the execution of each interaction complicated the testing process due to the unavailability of this latter, especially for some systems. Consequently, we propose, in this work, a new test approach based on mutation analysis, parallel genetic algorithms, a sequence diagram, and an activity diagram which can test each behavioural scenario individually. The aim is to pinpoint, in the case of error detection, the scenario that caused the detected error among the scenarios running in parallel.

Mutation analysis, proposed by [21], is the main technique used to validate and improve test data. It is based on fault injection (different types of programming errors). The first step of this technique is to create a set of erroneous versions of the program under test, called mutants (exact copy of the program under test in which a single simple error was injected), generated from the definition of a set of mutation operators automatically, and running a set of test cases on each of these faulty programs. The result of the execution of the test cases with the mutants allows, on the one hand, to evaluate the effectiveness of the set of test cases by the proportion of mutants detected; on the other hand, the analysis of the undetected errors helps to guide the generation of new test cases. This consists of covering the areas where these errors are located and generating specific test data to cover the particular cases. Mutation analysis was originally designed for testing procedural software, but for several years, many works have been interested in this technique in the context of object-oriented software [22–24]. For the MAS test, mutation analysis was used in two works. In the first work [25], the authors propose a testing approach based on mutation analysis to test the goals of an agent. In the second work [26], the authors propose error patterns to test the agent of the platform Jason.

The remainder of this paper is organised as follows. In Section 2, we provide a brief overview of major-related work. We describe, in Section 3, the proposed approach and its different phases. Section 4 illustrates the proposed approach using a concrete case study. Some conclusions and future work directions are given in Section 5.

2. Related Work

In recent years, only a few approaches have been proposed in the literature for testing multi-agent systems. We present, in what follows, some selected approaches:

De Wolf et al. [16]: This paper proposes an empirical analysis approach combining agent-based simulations and numerical algorithms for analysing the global behaviour of a self-organizing system. The initial values of macroscopic variables (those related to the properties that are studied) are supplied; a certain number of simulations are then initialised accordingly. Simulations are then executed for a predefined duration, and the average values of these variables on all the simulations are then given as results to the

analysis algorithm; this latter processes these results to compute the next initial values. The algorithm also decides on the configuration of the next simulations (initial conditions, durations), and the cycle is repeated until the algorithm reaches its goal (for instance, converge towards a value and find a steady state).

In Dehimi 2015 [17], we proposed a new model-based testing technique for holonic agents. The technique uses genetic algorithms and takes into account the evolution (successive versions) of an agent. The approach is organised into two main phases conducted iteratively. The first phase focuses on the detection of a new version of an agent under test. The second phase addresses the testing of each newly detected version. The new version of the agent is analysed in order to generate a behavioural model in which it is based on the generation of test cases. The test case generation process focuses on the new (and/or changed) parts of the agent behaviour. In this way, the technique supports an incremental update of the test cases, which is a crucial issue.

In Dehimi and Mokhati 2019 [18], we proposed a new test case generation approach based on the sequence diagram that could cover interactions between agents individually as well as possible scenarios that can be performed in an inclusive, exclusive, or parallel way. For this, we introduce plugs using the constraints, expressed in OCL, necessary for the execution of each interaction. The proposed approach is supported by a formal framework and applied to a concrete case study to illustrate and show its usefulness.

Thangarajah et al. [19] proposed an approach to quantify the goal's completeness in the BDI agent system. Completeness has been measured by considering resources consumed by a goal and measuring the effect of the goal in terms of the achieved desired outcomes. Factors that have been considered to quantify goal completeness include efforts, accomplishments, the number of actions performed by the agent, and the time taken for the action. Authors have taken the idea of a goal-plan tree, in which goals with relevant plans have been annotated to form a tree.

Bakar and Selamat [27] examined testing methods for agent systems in general, mapping the various properties and faults that existing techniques may detect. The objective was to identify research gaps and future research direction regarding agent systems verification.

Barnier et al. [28] compared software testing and multi-agent systems testing, more particularly in an embedded context. In this work, major multi-agent system testing techniques were analysed, with the AEIO facets for multi-agent systems. The objective was to provide a specific approach for testing MAS on embedded systems. The proposed approach was built on three basic levels: agent test, collective resources test, and acceptance test.

Winikoff [29] proposed a method for testability assessment on BDI (belief–desire–intention) agents, where two different adequacy criteria were compared: the all-edges criterion, which is satisfied if the set of tests covers all edges in a control-flow graph, and the all-paths criterion, where a test set is adequate if the test covers all paths in the control-flow graph of the program. The degree of testability obtained was used to indicate the number of test cases needed to validate a BDI program.

Gonçalves et al. [20] presented a whole framework for analysing and testing the MAS social level under the organisational model Moise+. This framework uses a Moise+ specifications set as an information artefact mapped in a coloured Petri net (CPN) model, named CPN4M, as a test case generation mechanism. CPN4M uses two different test adequacy criteria: all-paths and state-transition path. In this paper, we have formalised the transition from Moise+ to CPN, the procedures for test case generation, and executed some tests in a case study. The results indicate that this methodology can increase the correction degree for a social level in a multi-agent system specified by a Moise+ model, indicating the system context that can lead the MAS to failure.

Savarimuthu et al. [25]: This paper proposes a set of mutation operators for a cognitive agent-oriented programming language, specifically Goal. The proposed mutation operators are derived and guided systematically by an exploration of the bugs found in a collection of undergraduate programming assignments written in Goal. In fact, in exploring these

programs, the authors also provide an additional contribution: evidence of the extent to which the two foundational hypotheses of mutation testing hold for Goal's programs.

Huang et al. [26]: This paper applies SMT (semantic mutation testing) to three rule-based agent programming languages, namely Jason, GOAL, and 2APL, providing several contexts in which SMT is useful for these languages. It also proposes three sets of semantic mutation operators (i.e., rules to make semantic changes) for these languages, respectively, and a systematic approach to the derivation of semantic mutation operators for rule-based agent languages. This paper then shows that, through the preliminary evaluation of the proposed semantic mutation operators for Jason, SMT has some potential to assess the tests, robustness, and reliability of semantic changes.

Although these works have considerably forwarded the domain by proposing novel strategies for MAS testing, they avoided, on the one hand, considering the problem of testing a multi-agent system while having multiple behavioural scenarios running in parallel. On the other hand, using OCL constraints to solve this problem is not always valid due to their unavailability, especially for some systems. We present in this paper a new test case generation approach, essentially based on mutation analysis, parallel genetic algorithms, sequence diagrams, and activity diagrams, which is able to cover behavioural scenarios individually in a multi-agent system. This is to pinpoint, in the case of the detection of an error, the scenario that caused the detected error among the scenarios running in parallel.

3. The Proposed Approach

The proposed approach consists of generating test cases, composed of test case inputs and test case outputs, that can cover each behavioural scenario individually. The aim is to identify exactly, in the case of error detection, the scenario that originated the detected error, among the scenarios running in parallel. The approach consists of three phases (Figure 1), namely:

The first phase consists of generating, for each scenario S_i, the test case inputs $Inputs_i$ with which we must be sure that the agents of the system execute the scenario S_i with only, namely, the set of its interactions Int_j and the set of its activities $Activitys_j$ triggered by those interactions. This phase uses the technique of mutation analysis. It consists of generating, for each interaction, a mutant, where each generated mutant $Mutant_j$ associated with the interaction Int_j represents an exact copy of the system in which only one error $Error_j$ is injected at the level of the interaction Int_j. The generation of mutants facilitates the generation of inputs that ensure the execution of each S_i scenario individually. Indeed, knowing the detection of the error $Error_j$ injected into the interaction Int_j during the execution of mutant $Mutant_j$ by given inputs means that these inputs allowed the execution to pass through the interaction Int_j. This means that to find the inputs which ensure the execution of a given scenario S_i individually, it suffices to find, the inputs allowing to detect during the execution, all the errors of the mutants associated with the interactions of this scenario S_i. These inputs should not detect errors of other mutants associated with interactions of other scenarios. To identify these inputs, this phase uses parallel genetic algorithms, where each of these algorithms A_i is responsible for finding the inputs $Inputs_i$ that ensure the execution of the scenario S_i only. For this, each algorithm must find among a set of given inputs those which best satisfy its fitness function F_Si which is defined as follows:

$$F\_Si = A/B + A/C$$

where:

- A is the number of killed mutants associated with S_i
- B is the number of killed mutants
- C is the total number of mutants associated with S_i

The satisfaction of the fitness function F_S_i, generation after generation, means that the algorithm A_i (responsible for finding the inputs $Inputs_i$ that ensure the execution of the scenario S_i only) is progressively becoming closer to identifying the inputs that can detect

the maximum number of errors injected into the interactions of this scenario, in addition to the minimum number of errors injected into the interactions of other scenarios. The fitness function F_S_i is completely satisfied if the algorithm A_i locates the inputs that can detect all the errors injected into the interactions of this scenario without the detection of any errors injected into the interactions of any other scenarios. These inputs represent the desired inputs of the test cases that ensure covering the scenario S_i only.

The second phase consists of finding the expected outputs $Outputs_i$ for the inputs $Inputs_i$ generated in the first phase. This phase uses the activity diagram to determine the activities that will be triggered by the interactions of each scenario. This facilitates the generation of test case outputs. Indeed, for this, it suffices to apply to the generated inputs $Inputs_i$, for each scenario S_i, the set of activities $Activitys_i$ that will be triggered by the interactions of this latter. Here, it is assumed that these activities are tested in the unit level test, and they do not contain errors.

The third phase aims to detect possible errors in each scenario. For a given scenario S_i, the detection of errors (injected into its interactions to generate mutants) using $Intputs_i$ (generated in the first phase for the scenario S_i using parallel genetic algorithms), ensures that the execution passes through this scenario only, but it does not ensure that the scenario is errorless. To detect real errors presented in a given scenario S_i, this phase consists of running the original system with the inputs $Intputs_i$ and comparing the obtained results with the expected outputs $Outputs_i$ (generated in the second phase for the same scenario S_i). Any difference (more or less) between the obtained outputs and the expected outputs means that there are errors in the scenario S_i. The errors detected are surely included in the scenario S_i under test because the $Inputs_i$ used for the test of this scenario assured us that it was executed individually.

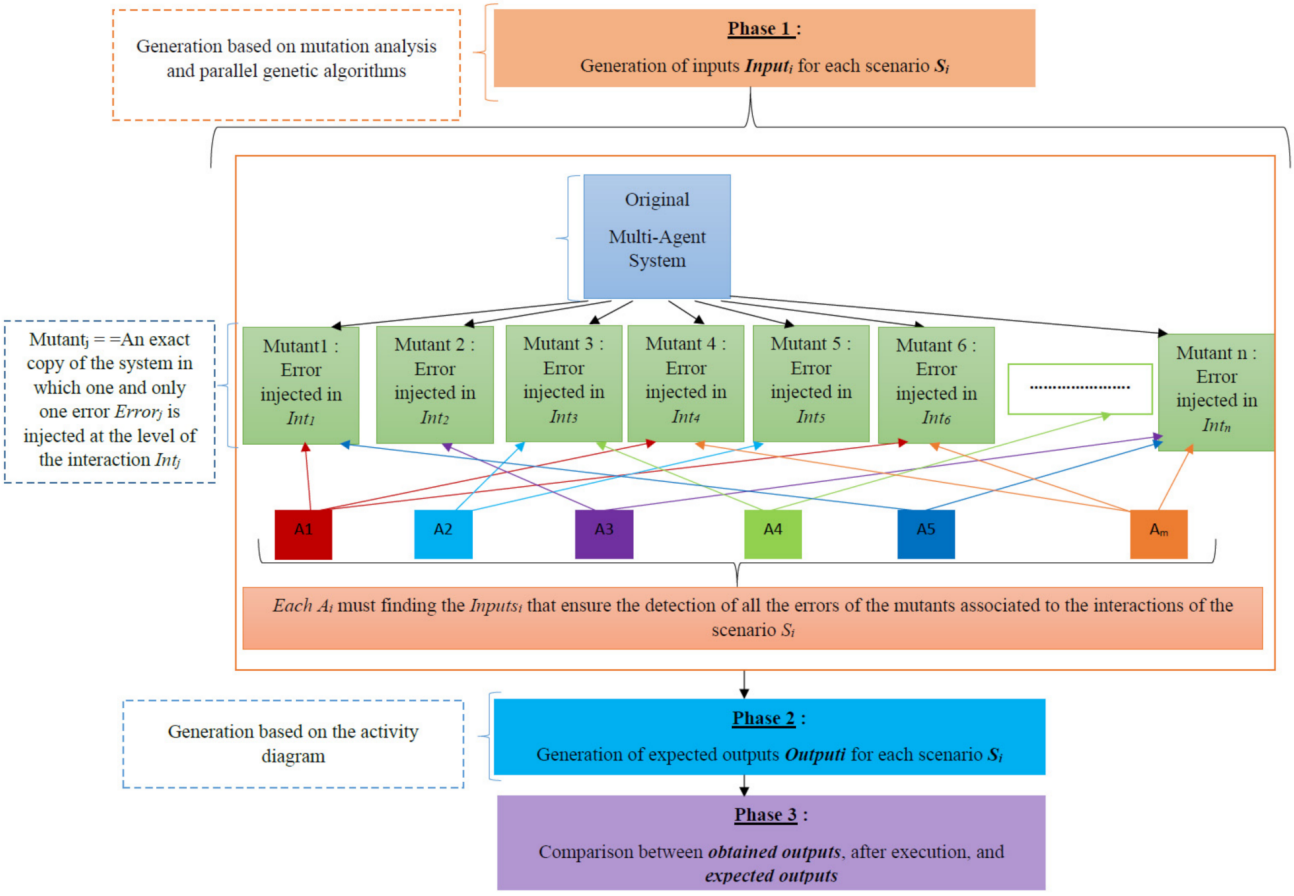


Figure 1. The phases of the proposed approach.

In order to automate the phases of our approach, we have used the sequence diagram and the activity diagram to generate a graph G. In this graph, each path P_i represents a scenario S_i for which we must find the inputs $Inputs_i$ that ensure the execution of the interactions of this scenario only and the outputs $Outputs_i$ expected for these inputs. Each node n_j in the graph G represents an interaction between two agents in the sequence diagram. It contains the necessary information for the generation of the inputs of the test case, namely, the error to be injected into the interaction Int_j represented by this node to obtain a mutant $Mutant_j$ (incorrect version of the system at the level of the interaction Int_j). It also contains the information necessary for the generation of the expected outputs for the generated inputs, namely, the activities to be performed by the receiving agent following the receipt of the interaction Int_j represented by the node n_j. In the following, we will present a formal definition of the proposed graph as well as the processes of the three phases of the proposed approach.

3.1. Formal Definition of the Proposed Graph G

According to our approach, the graph G is defined as follows: G = $\{P_i\}/_{0 \leq i \leq N}$ where:

- P_i represents a path i in the graph associated with the scenario S_i. Each path P_i is defined as follows: $P_i = \{n_0, n_i, n_f\}$ where:
- n_0 is the initial node representing a state from which an operation begins (the precondition of the scenario S_i corresponding to P_i).
- n_f is the final node representing a state where an operation end (the post-condition of the scenario S_i corresponding to P_i).
- n_i is the set of all the nodes of the path P_i where each node $n_j/_{0 \leq j \leq Mi}$ belongs to n_i is defined as follows: $n_j = <Int_j, Activitys_j, Error_j>$ where:
  - Int_j represents the interaction represented by node n_j. It is defined as follows: Int_j = {m, from_Agent, to_Agent} where:
    - m is the name of the message with its signature,
    - from_Agent is the sender of the message,
    - to_Agent is the recipient of the message.
  - $Activitys_j$: represents all the activities to be performed following the interaction Int_j by the receiving agent. Each $Activity_k$ belonging to $Activitys_j$ is defined as follows: $Activity_k = < Inputs_k, Outputs_k, Operation_k>$ where:
    - $Inputs_k$: represents the inputs of activity k.
    - $Outputs_k$: represents the outputs of activity k.
    - $Operation_k$: represents the operations of activity k which will be executed on $Inputs_k$ to obtain $Outputs_k$.
  - $Error_j$: represents the error to be introduced at the level of the interaction Int_j to obtain the mutant $Mutant_j$.

3.2. Generation of Test Case Inputs

The generation of the inputs of the test case begins with the generation of mutants. For that, it is enough to go through all of the graph G, node by node, and for each node n_j, a version of the system is generated, in which the error $n_j.error_j$ associated with this node is injected at the level of the interaction Int_j. Algorithm 1 summarises the generation of the mutants' process.

Algorithm 1: Mutants_Generation

Inputs: The graph G

Outputs: Mutants

1 **Begin**

2 **For** each node $n_j \in$ G do

3 $Mutant_j$ = An exact copy of the system;

4 Inject n_j. Error in the $Mutant_j$

5 **End For;**

6 **End.**

After the generation of mutants, we move on to the generation of the test case inputs for each scenario S_i using parallel genetic algorithms (Figure 2). To do this, we start with the generation of an initial population Pop_0, in which a large number of individuals is constructed. Each individual of this population is represented as an input vector V_k. The generation of the initial population is performed randomly.

From the initial population, each algorithm A_i begins to search for the individuals (the inputs vectors) that satisfy its fitness function F_S_i. In order to find a more efficient population Pop_1 that can best meet its fitness function, each algorithm applies the following steps to the Pop_0:

- First, each individual V_k of the population Pop_0 is evaluated. For this, the value of the fitness function F_S_i is calculated.
- Then, a selection step is applied. This step eliminates the worst individuals from Pop_0 and keeps only the best ones according to their evaluation. Here, according to the formula of the fitness function F_S_i, the individuals (the vectors of inputs) kept are characterised by the possibility of killing the maximum number of mutants associated with the interactions of the scenario S_i and the minimum number of mutants associated with the interactions of the other scenarios.
- The next step is to crossbreed the previously selected individuals to obtain the new Pop_1 population. For this, every two individuals (parents) will be crossed in order to obtain a new individual (descendant). The latter is composed of part of parent 1 and part of parent 2 (*one-point crossover*).
- To diversify the solutions over the generations, a mutation step is used. This mutation consists of modifying a small part in certain individuals of the new generation randomly.

From the obtained population Pop_1, the steps described above will be reapplied within the limit of the number of possible generations. This is in order to obtain a population in which there are individuals (input's vectors) who fully satisfy the fitness function. If these individuals were not obtained within the limit of possible generations, the best individuals obtained in the last generation (Pop_n) will be taken. Figure 2 summarises the test case inputs generation process.

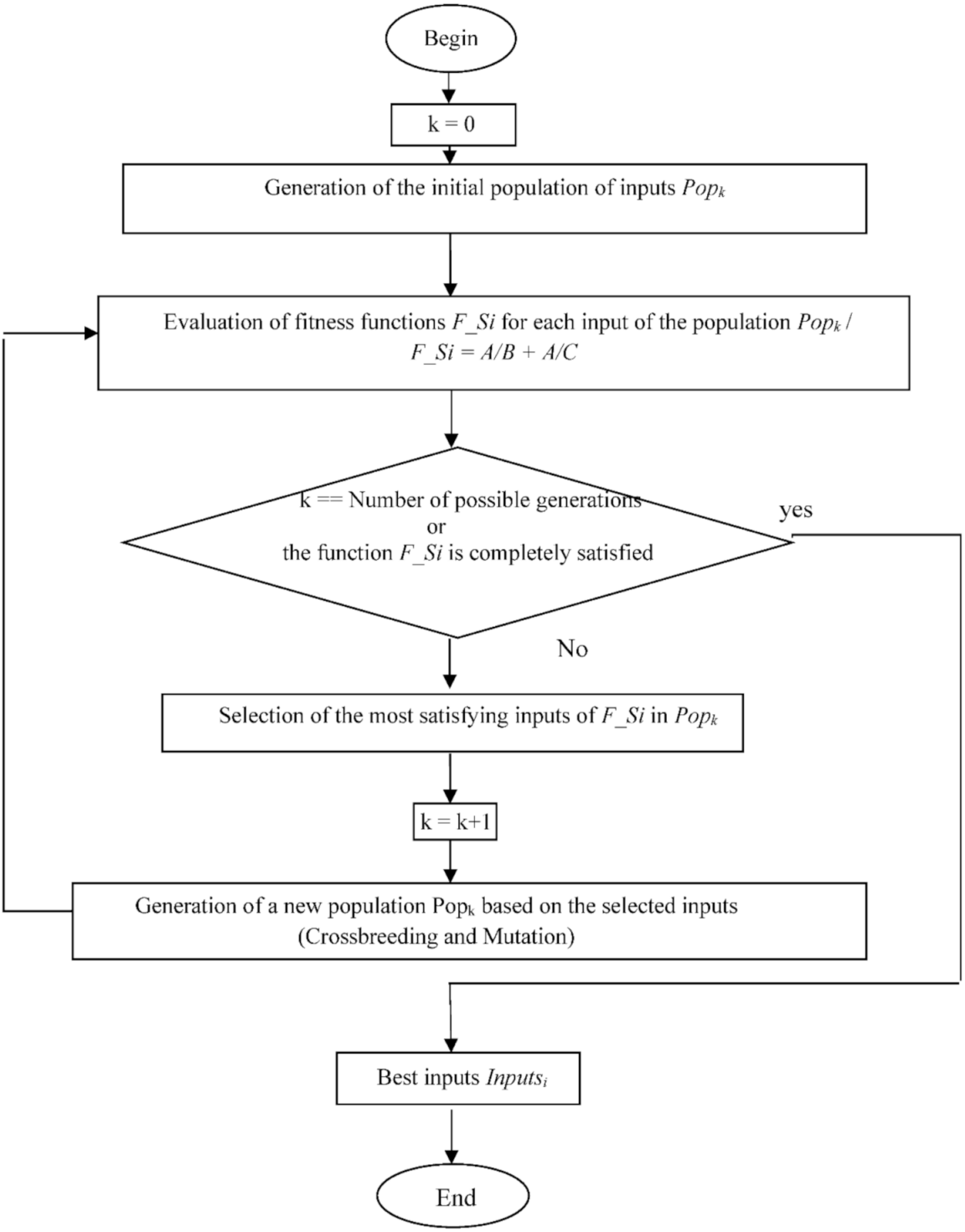


Figure 2. Test case inputs generation process.

3.3. Generation of the Expected Test Case Outputs

For the generation of the expected outputs $Outputs_i$ for the inputs $Inputs_i$ of the scenario S_i, it suffices to go through all the paths P_i of the graph G node by node, and for each node n_j of the path P_i, we calculate the results of the application of the activities $n_j.Activitys_j$ of this node on the inputs $Inputs_i$. Algorithm 2 summarises the generation of the test case outputs process.

Algorithm 2: Expected_Outputs_Generation

Inputs: The graph G

Outputs: Expected Outputs

Begin

======= 1 Generate all paths $P = \{p_1, p_2, \ldots\ldots, p_N\}$ from the start nod ==== to the final node in the graph G

================ 2 Generation of Outputs ==============

For each path $P_i \in P$ **do**

$Outputs_i = \Phi$;

For each $n_j \in P_i$

$Outputs_j = \Phi$

For each $Activity_k \in n_j.Activity_j$

$Outputs_k = Operation_k(Inputs_k)$

$Outputs_j = Outputs_j \cup Outputs_k$

End For;

End For;

$Outputs_i = Outputs_i \cup Outputs_j$

End For;

End.

3.4. Detection of Errors

For the detection of errors, it is sufficient to compare, for each scenario S_i, the expected outputs $outputs_i$ and the outputs obtained after the execution of the original system. The non-conformance between the obtained results and the expected outputs implies that there is an interaction error and non-conformance between the obtained post-condition and the expected one, which implies that there is a scenario error. An interaction error can be a bad reply to a message, a correct message sent to an incorrect agent, or an incorrect message sent to the correct agent, an invocation message with inappropriate or incorrect arguments, wrong or missing output, and so on. However, a scenario error can be an abnormal shutdown, a sequence of messages that do not follow the desired trajectory, and so on. Figure 3 summarises the errors' detection process.

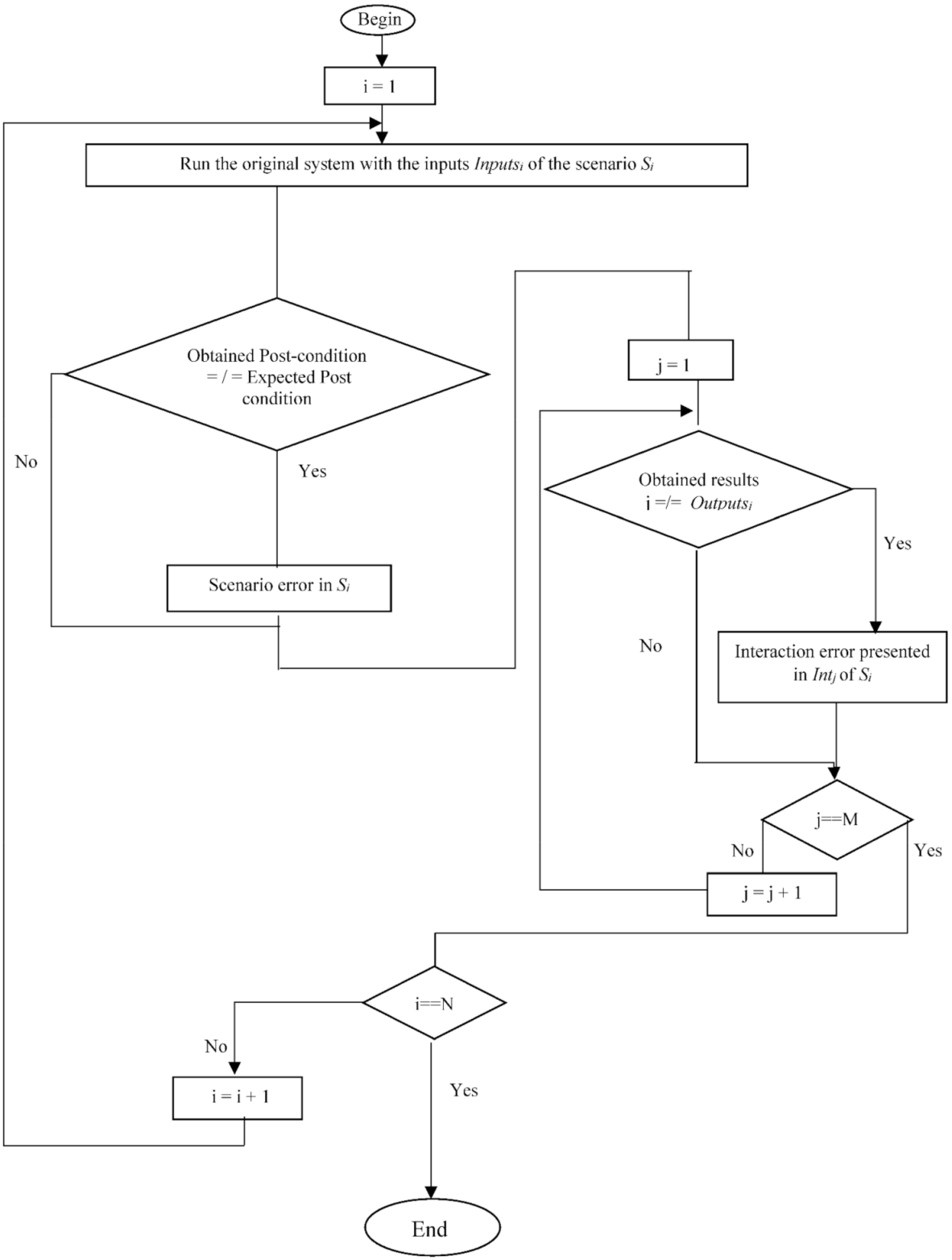


Figure 3. Error detection process.

4. Case Study

To validate our approach, we apply it to a concrete case study that will represent the interest of the proposed approach. The system chosen for this study is an Airport management system. The purpose of this system is to provide management at an airport by automatically assigning a pilot as well as a plane to a request, taking into account several variants such as the quantity to be transported as well as the type of transport, which influences the category of the plane, the final destination, which influences the choice of the pilot, and the availability of planes and pilots.

The airport management system is composed of several interacting agents, namely:

- Agent MainGui: This is an interface agent which aims to receive, control, and verify warning requests from the customer; it is considered an intermediary between the user and the system.
- Agent Plane: This is the agent responsible for all categories of planes and its job is to determine which agent (Agent Heavy Plane Commodity, Agent Light Plane Commodity or Agent Civil) is most appropriate to handle the request. For this, it must verify the quantity requested as well as the type of transport required, and then it will transmit the request to the agent in question.
- Agent Heavy Plane Commodity: Responsible for processing requests for heavy commodity planes.
- Agent Light Plane Commodity: Responsible for processing requests for light commodity planes.
- Agent Civil: Responsible for processing requests for planes that carry civilians; the quantity here will serve as the number of passengers on the plane.
- Agent Pilot: This is the agent responsible for all pilots and its job is to determine, depending on the city of destination, which agent (Agent Pilot Europe or Agent Pilot Local) is most appropriate to handle the request.
- Agent Pilot Europe: Responsible for processing requests for destinations in Europe.
- Agent Pilot Local: Responsible for processing requests for local destinations only.

Depending on the overall quantity to be transported (Qo $\geq$ 1000 or Qo < 1000) and the type of transport (Commodity or Civil), Agent Plane sends a request to specialised light, heavy, or civilian plane agents. These agents, depending on the availability of the planes that they manage, can accept or reject requests. In the event that (Qo = 0), a refusal message will be returned immediately. At the same time, depending on the European or local destination zone, the Agent Pilot agent sends a request to the agents: Agent Pilot Europe and/or Agent Pilot Local. These agents, depending on the availability of their resources, can accept or reject the request. In the event that the destination is not supported, a rejection message will be sent immediately.

Once the requested quantity is properly assigned to the planes and pilots, the agents, Agent Plane and Agent Pilot, each establish a transportation plan. The transportation plan includes the following elements:

- The request ID.
- A Plane object containing information specific to the affected plane.
- A Pilot object containing information specific to the affected pilots.
- A trace of the request path from the sender to the receiver.
- Unique messages are exchanged to differentiate the type of response request between agents.
- Everything is encapsulated in a Response Plan for each pilot/plane.

These plans will then be merged to return a response plan to the query knowing that in the case of failure, the plans may be empty.

In the following, we will describe the different phases of our approach necessary for the test of this system, namely, the generation of the graph G, the generation of the inputs of the test cases, the generation of the expected outputs of the test cases, and the detection of errors. The results shown in each phase are obtained automatically following the use of a software tool that we have developed.

4.1. Generation of the Graph G

The generation of the graph G represents the first step in the process of our approach. It consists of transforming automatically the sequence diagram (Figure 4) and the activity diagram (Figure 5) into a graph. The generation of the graph goes through an intermediate step, which consists of transforming the sequence diagram and the activity diagram into an XML file. The purpose of this is to simplify the creation and the filling of the various nodes of the graph.

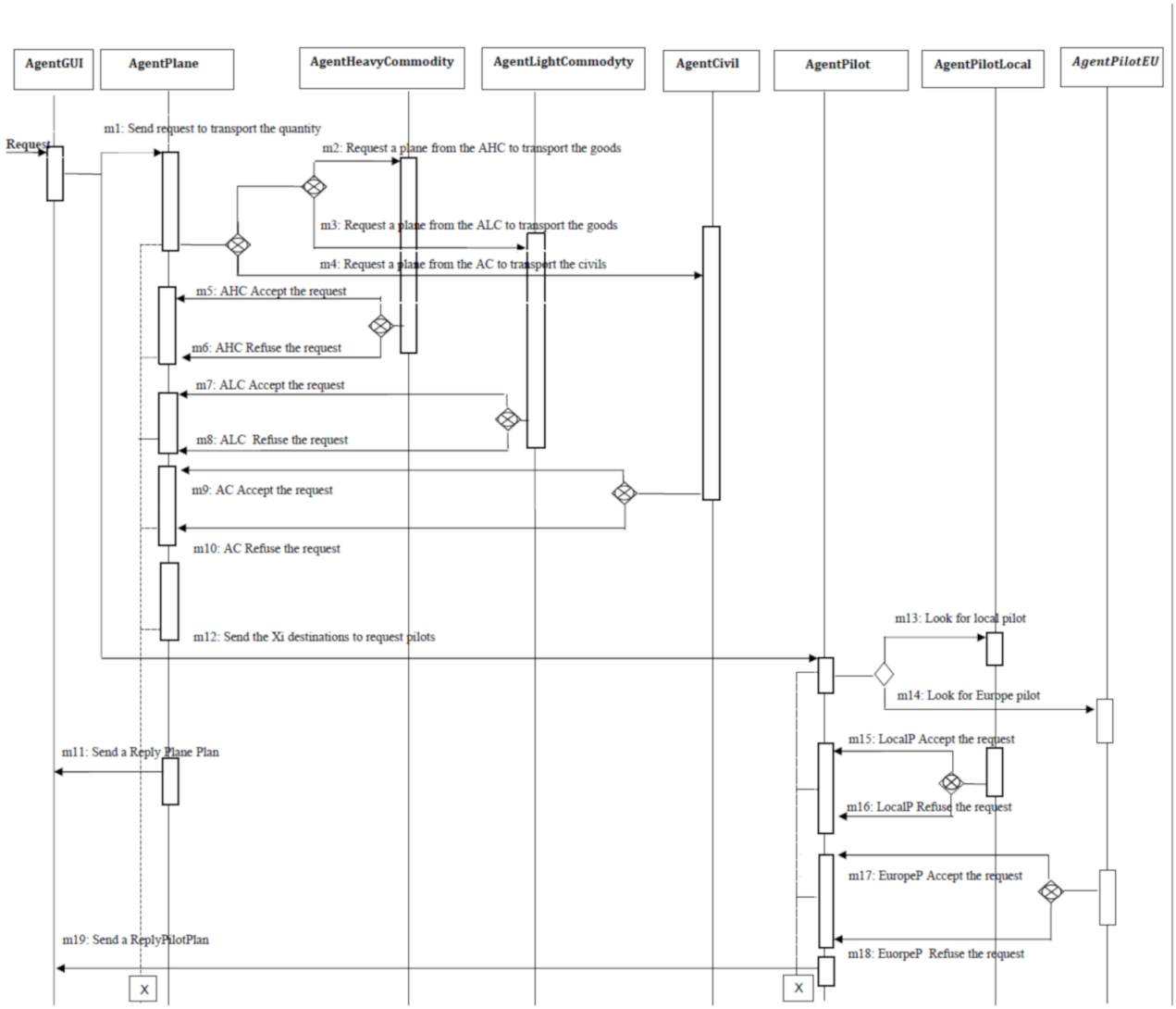


Figure 4. Sequence diagram.

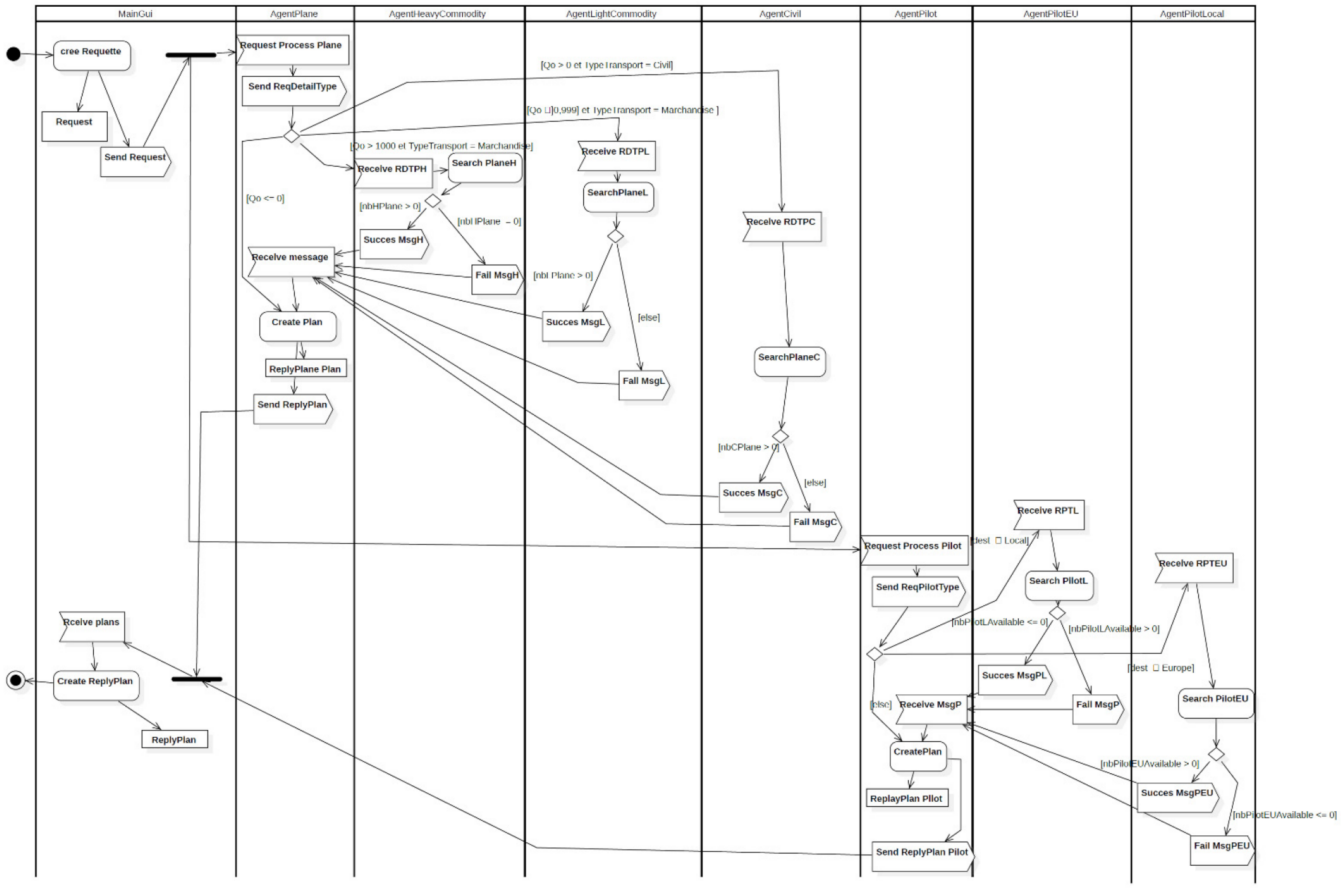


Figure 5. Activity diagram.

The graph G, obtained following the transformation of the XML file, is presented in Figure 6. It contains 19 nodes, namely, (i) node 0, which represents the initial node n_0, (ii) nodes 11 and 19, which represent final nodes n_f, (iii) and the other nodes, presented in blue, which represent interaction nodes n_j. Each interaction node n_j contains the following information: the identifier of the interaction, the sender and the receiver of the interaction, the message exchanged, the activities triggered by this interaction, and finally, the error that will be injected into this interaction to generate the corresponding mutant.

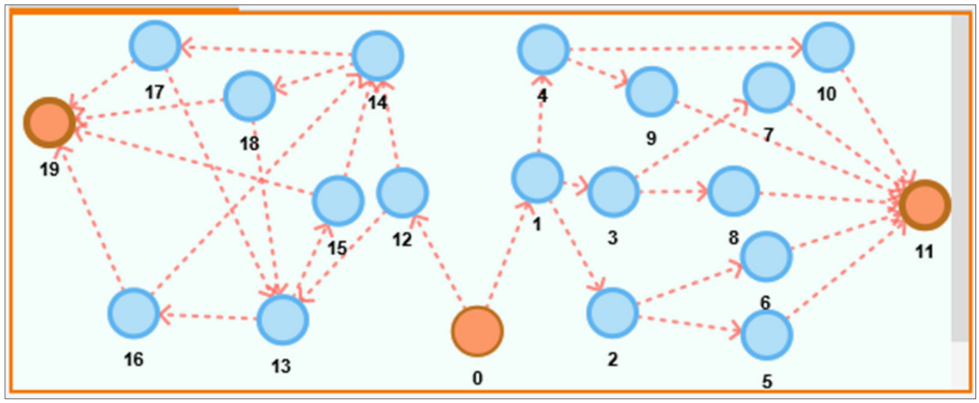


Figure 6. The obtained graph G.

Each path P_i of the graph G represents a possible behaviour scenario S_i for the system under test. The generation of the P_i paths led us to 14 paths, in total, as shown in Table 1. Here, the detail of the information stored in the nodes is presented.

Table 1. Graph paths P_i (*System scenarios* S_i).

Paths P_i (Scenarios S_i)	**The Nodes of the Generated Paths**				
P1	Initial	N1 :((AgentGui, Aplane), MainGuiToAPlane, Action1, Error//1).	N2 :((AgentPlane, AgentHeavyCommodity), APlaneToAPlaneH, Action2, Error//2).	N6 :((AgentPlane, AgentHeavyCommodity, AgentPlane), APlaneHToAPlaneF, Action6, Error//6).	N11 :((AgentPlane, AgentGui), APlaneToMainGui, Action11, Error//11) Final_1.
P2	Initial	N1 :((AgentGui, Aplane), MainGuiToAPlane, Action1, Error//1).	N2 :((AgentPlane, AgentHeavyCommodity), APlaneToAPlaneH, Action2, Error//2).	N5 :((AgentPlane, AgentHeavyCommodity, AgentPlane), APlaneHToAPlaneI, Action5, Error//5).	N11 :((AgentPlane, AgentGui), APlaneToMainGui, Action11, Error//11) Final_1.
P3	Initial	N1 :((AgentGui, Aplane), MainGuiToAPlane, Action1, Error//1).	N3 :((AgentPlane, AgentLightCommodity), APlaneToAPlaneL, Action3, Error//4).	N8 :((AgentPlane, AgentLightCommodity, AgentPlane), APlaneLToAPlaneF, Action8, Error//8).	N11 :((AgentPlane, AgentGui), APlaneToMainGui, Action11, Error//11) Final_1.
P4	Initial	N1 :((AgentGui, Aplane), MainGuiToAPlane,Action1, Error//1).	N3 :((AgentPlane, AgentLightCommodity), APlaneToAPlaneL, Action3, Error//4).	N7 :((AgentPlane, AgentLightCommodity, AgentPlane), APlaneLToAPlaneI, Action7, Error//7).	N11 :((AgentPlane, AgentGui), APlaneToMainGui, Action11, Error//11) Final_1.
P5	Initial	N1 :((AgentGui, Aplane), MainGuiToAPlane,Action1, Error//1).	N4 :((AgentPlane, AgentCivil), APlaneToAPlaneC, Action4, Error//4).	N10 :((AgentCivil, AgentPlane), APlaneCToAPlaneF, Action10, Error//10).	N11 :((AgentPlane, AgentGui), APlaneToMainGui, Action11, Error//11) Final_1.
P6	Initial	N1 :((AgentGui, Aplane), MainGuiToAPlane, Action1, Error//1).	N4 :((AgentPlane, AgentCivil), APlaneToAPlaneC, Action4, Error//4).	N9 :((AgentCivil, AgentPlane), APlaneCToAPlaneI, Action9, Error//9).	N11 :((AgentPlane, AgentGui), APlaneToMainGui, Action11, Error//11) Final_1.
P7	Initial	N12 :((AgentGui, AgentPilot), MainGuiToAPilot, Action12, Error//12).	N13 :((AgentPilot, AgentPilotEU), APilotToAPilotEU, Action13, Error//13).	N16 :((AgentPilotEU, AgentPilot), APilotEUToAPilotF, Action16, Error//16).	N19 :((AgentPilot, AgentGui), APilotToMainGui, Action19, Error//19) Final_2.

Table 1. *Cont.*

Paths P_i (Scenarios S_i)		**The Nodes of the Generated Paths**					
P8	Initial	N12 :((AgentGui, AgentPilot), MainGui-ToAPilot, Action12, Error//12).	N13 :((AgentPilot, AgentPilotEU), APilotToAPilotEU, Action13, Error//13).	N16 :((AgentPilotEU, AgentPilot), APilo-tEUToAPilotF, Action16, Error//16).	N14 :((AgentPilot, AgentPilotLocal), APilotToAPilotL, Action14, Error//14).	N18 :((Agent-PilotLocal, AgentPilot), APilotLToAPi-lotF, Action18, Error//18).	N19 :((AgentPilot, AgentGui), APilotToMain-Gui, Action19, Error//19) Final_2.
P9	Initial	N12 :((AgentGui, AgentPilot), MainGui-ToAPilot, Action12, Error//12).	N13 :((AgentPilot, AgentPilotEU), APilotToAPilotEU, Action13, Error//13).	N16 :((AgentPilotEU, AgentPilot), APilo-tEUToAPilotF, Action16, Error//16).	N14 :((AgentPilot, AgentPilotLocal), APilotToAPilotL, Action14, Error//14).	N17 :((Agent-PilotLocal, AgentPilot), APilotLToAPi-lotI, Action17, Error//17).	N19 :((AgentPilot, AgentGui), APilotToMain-Gui, Action19, Error//19) Final_2.
P10		N12 :((AgentGui, AgentPilot), MainGui-ToAPilot, Action12, Error//12).	N13 :((AgentPilot, AgentPilotEU), APilotToAPilotEU, Action13, Error//13).	N15 :((AgentPilotEU, AgentPilot), APilo-tEUToAPilotI, Action15, Error//15).	N19 :((AgentPilot, AgentGui), APi-lotToMainGui, Action19, Error//19) Final_2.		
P11	Initial	N12 :((AgentGui, AgentPilot), MainGui-ToAPilot, Action12, Error//12).	N13 :((AgentPilot, AgentPilotEU), APilotToAPilotEU, Action13, Error//13).	N15 :((AgentPilotEU, AgentPilot), APilo-tEUToAPilotI, Action15, Error//15).	N14 :((AgentPilot, AgentPilotLocal), APilotToAPilotL, Action14, Error//14).	N18 :((Agent-PilotLocal, AgentPilot), APilotLToAPi-lotF, Action18, Error//18).	N19 :((AgentPilot, AgentGui), APilotToMain-Gui, Action19, Error//19) Final_2.
P12	Initial	N12 :((AgentGui, AgentPilot), MainGui-ToAPilot, Action12, Error//12).	N13 :((AgentPilot, AgentPilotEU), APilotToAPilotEU, Action13, Error//13).	N15 :((AgentPilotEU, AgentPilot), APilo-tEUToAPilotI, Action15, Error//15).	N14 :((AgentPilot, AgentPilotLocal), APilotToAPilotL, Action14, Error//14).	N17 :((Agent-PilotLocal, AgentPilot), APilotLToAPi-lotI, Action17, Error//17).	N19 :((AgentPilot, AgentGui), APilotToMain-Gui, Action19, Error//19) Final_2.
P13	Initial	N12 :((AgentGui, AgentPilot), MainGui-ToAPilot, Action12, Error//12).	N14 :((AgentPilot, AgentPilotLo-cal), APilotToAPi-lotL, Action14, Error//14).	N18 :((AgentPilotLocal, AgentPilot), APilotLToAPilotF, Action18, Error//18).	N19 :((AgentPilot, AgentGui), APi-lotToMainGui, Action19, Error//19) Final_2.		
P14	Initial	N12 :((AgentGui, AgentPilot), MainGui-ToAPilot, Action12, Error//12).	N14 :((AgentPilot, AgentPilotLo-cal), APilotToAPi-lotL, Action14, Error//14).	N17 :((AgentPilotLocal, AgentPilot), APilotLToAPilotI, Action17, Error//17).	N19 :((AgentPilot, AgentGui), APi-lotToMainGui, Action19, Error//19) Final_2.		

4.2. Generation of Mutants and Test Case Inputs

4.2.1. Generations of Mutants

The generation of mutants is necessary for the generation of the test case inputs. It consists of applying the algorithm of the generation of mutants, mentioned previously, on the generated graph G. The algorithm generates for each node n_j of the graph an exact copy of the system into which it injects, at the level of the interaction corresponding to this node, the error stored in this latter. The injected error will be a modification in the message of the interaction in question. In fact, the erroneous interaction message becomes the following form:

Message + // + Error + // + « id » where Message represents the original message of the interaction and Error + id represents the change made.

In our case, the graph contains "19" nodes which bring us to "19" mutants; each, according to its identifier, will have an erroneous interaction; for example, mutant "2" will have the message of the interaction "2": ***APlaneToAHC*** modified to ***APlaneToAHC//Error//2.***

4.2.2. Generation of Test Case Inputs

The generation of test case inputs consists of finding, for each scenario S_i, the inputs $Inputs_i$ which ensure the execution of this scenario individually. It starts by generating randomly an initial population Pop_0 of an input's vectors where each vector V_k is defined as follows: V_k = (*Quantity*, *Transport Type* (*Merchandise/Civil*), *Destination Town* (*European/Local*), *Planes* (*Heavy Planes* (*name*, *companyName*, *transportSize*, *isFree* (*true*, *false*))/*Light Planes*(*name*, *companyName*, *transportSize*, *isFree* (*true*, *false*))/*Civil Planes*(*name*, *companyName*, *transportSize*, *isFree* (*true*, *false*))), *Pilots* (*Pilots Local* (*name*, *companyName*, *workingDest*, *isFree* (*true*, *false*))/*Pilots Europe* (*name*, *companyName*, *workingDest*, *isFree* (*true*, *false*)).

From the initial population, each genetic algorithm A_i, responsible for finding the inputs $Inputs_i$ of the scenario S_i, reapplies on all input vectors of each generation a set of steps presented previously. This is performed until the desired test case inputs $Inputs_i$ are obtained. The desired test case inputs are those that have a high satisfaction rate for the fitness function. In fact, the higher the rate, the more accurate the results are because if the rate is 100%, this means that the inputs $Inputs_i$ of the test cases ensure the execution of the scenario S_i only, and they will not trigger any other system interaction. If the rate is not 100%, this means that there are some interactions that do not belong to the scenario S_i but are still in execution.

The results of the application of the parallel genetic algorithms are presented in the form of a curve in the table (Table 2), where the pinnacles represent inputs that will fully satisfy the fitness function *F_Si* of the scenario S_i. The values of some of these inputs are presented with their expected outputs in Table 3 of the following section.

Table 2. Obtained results for the inputs of the test cases.

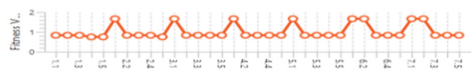
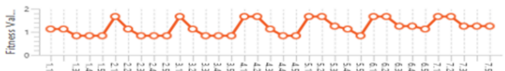
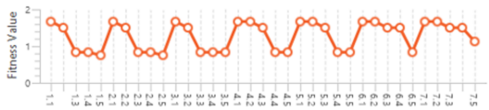
Si	Curves	Si	Curves
S1		S8	
S2		S9	
S3		S10	

Table 2. *Cont.*

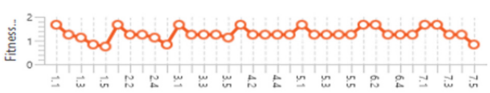
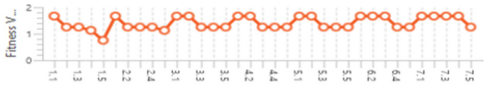
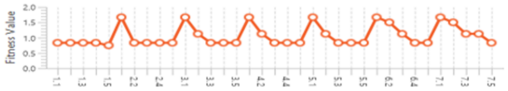
Si	Curves	Si	Curves
S4		S11	
S5		S12	
S6		S13	
S7		S14	

Table 3. Results of the generation of test cases (*inputs and expected outputs*) *obtained for some scenarios*.

S_i	$Inputs_i$	$Outputs_i$
S1	ID: d4b1be08-e62f-44da-bb88-33f0bf6c00b0 Quantity: 1273 Transport Type: Merchandise Destination Town: [Ville [name=, pays=, location=Asie]] Planes: Heavy Planes: [Plane [name=plane, companyName=company, transportSize=1500, tt=Merchandise, isFree=false]] Light Planes: [Plane [name=plane, companyName=company, transportSize=999, tt=Merchandise, isFree=true]] Civil Planes: [Plane [name=plane, companyName=company, transportSize=100, tt=Civil, isFree=true]] Pilots: Pilots Local: [Pilot [name=pilot, companyName=company, workingDest=Local, isFree=true]] Pilots Europe: [Pilot [name=pilot, companyName=company, workingDest=Europe, isFree=true]] Fitness Value: 2 Percent:100%	ID: d4b1be08-e62f-44da-bb88-33f0bf6c00b0 Trace:[MainGuiToPlane, APlaneToAHC, AHCToAPlaneF, APlaneToMainGui] Plane: null Pilot: []
S2	ID: 957b7f77-f8bd-4ff9-8ad7-75affcdb9ce2 Quantity: 1310 Transport Type: Merchandise Destination Town: [Ville [name=, pays=, location=Asie]] Planes: Heavy Planes: [Plane [name=plane, companyName=company, transportSize=1500, tt=Merchandise, isFree=true]] Light Planes: [Plane [name=plane, companyName=company, transportSize=999, tt=Merchandise, isFree=true]] Civil Planes: [Plane [name=plane, companyName=company, transportSize=100, tt=Civil, isFree=true]] Pilots: Pilots Local: [Pilot [name=pilot, companyName=company, workingDest=Local, isFree=true]] Pilots Europe: [Pilot [name=pilot, companyName=company, workingDest=Europe, isFree=true]] Fitness Value: 2 Percent:100%	ID: 957b7f77-f8bd-4ff9-8ad7-75affcdb9ce2 Trace:[MainiToPlane,APlaneToAHC,AHCToAPlaneI, APlaneToMainGui] Plane: Plane [name=plane, companyName=company, transportSize=1500, tt=Merchandise, isFree=false] Pilot: []

Table 3. *Cont.*

S_i	$Inputs_i$	$Outputs_i$
S5	ID: 6ee25abd-dabd-4f75-b923-73a1354ab493 Quantity: 21 Transport Type: Civile Destination Town: [Ville [name=ville5, pays=pays5, location=Amerique]] Planes: Heavy Planes: [Plane [name=plane1, companyName=Company, transportSize=1401, tt=Merchandise, isFree=false], Plane [name=plane2, companyName=Company, transportSize=1349, tt=Merchandise, isFree=false]] Light Planes: [Plane [name=plane1, companyName=Company, transportSize=789, tt=Merchandise, isFree=false], Plane [name=plane2, companyName=Company, transportSize=561, tt=Merchandise, isFree=true], Plane [name=plane3, companyName=Company, transportSize=802, tt=Merchandise, isFree=true], Plane [name=plane4, companyName=Company, transportSize=568, tt=Merchandise, isFree=false]] Civil Planes: [Plane [name=plane1, companyName=Company, transportSize=27, tt=Civile, isFree=false]] Pilots: Pilots Local: [Pilot [name=pilot0, companyName=company, workingDest=Local, isFree=false]] Pilots Europe: [Pilot [name=pilot0, companyName=company, workingDest=Europe, isFree=true]] Fitness Value: 2 Percent:100%	ID: 6ee25abd-dabd-4f75-b923-73a1354ab493 Trace:[MainGuiToPlane, APlaneToAC, ACToAPlaneF, APlaneToMainGui] Plane: null Pilot: []
S6	ID: 5b5ac228-693e-4477-aa59-5828bc4c9265 Quantity: 69 Transport Type: Civile Destination Town: [Ville [name=ville5, pays=pays5, location=Asie]] Planes: Heavy Planes: [Plane [name=plane1, companyName=Company, transportSize=1286, tt=Merchandise, isFree=false], Plane [name=plane2, companyName=Company, transportSize=1272, tt=Merchandise, isFree=true]] Light Planes: [Plane [name=plane1, companyName=Company, transportSize=952, tt=Merchandise, isFree=true]] Civil Planes: [Plane [name=plane1, companyName=Company, transportSize=46, tt=Civile, isFree=false], Plane [name=plane2, companyName=Company, transportSize=61, tt=Civile, isFree=false], Plane [name=plane3, companyName=Company, transportSize=84, tt=Civile, isFree=true], Plane [name=plane4, companyName=Company, transportSize=12, tt=Civile, isFree=true], Plane [name=plane5, companyName=Company, transportSize=84, tt=Civile, isFree=false], Plane [name=plane6, companyName=Company, transportSize=7, tt=Civile, isFree=true], Plane [name=plane7, companyName=Company, transportSize=25, tt=Civile, isFree=false], Plane [name=plane8, companyName=Company, transportSize=22, tt=Civile, isFree=false]] Pilots: Pilots Local: [Pilot [name=pilot0, companyName=company, workingDest=Local, isFree=false], Pilot [name=pilot1, companyName=company, workingDest=Local, isFree=true], Pilot [name=pilot2, companyName=company, workingDest=Local, isFree=true], Pilot [name=pilot3, companyName=company, workingDest=Local, isFree=true]] Pilots Europe: [Pilot [name=pilot0, companyName=company, workingDest=Europe, isFree=false], Pilot [name=pilot1, companyName=company, workingDest=Europe, isFree=false], Pilot [name=pilot2, companyName=company, workingDest=Europe, isFree=true]] Fitness Value: 2 Percent:100%	ID: 5b5ac228-693e-4477-aa59-5828bc4c9265 Trace:[MainGuiToPlane, APlaneToAC, ACToAPlaneI, APlaneToMainGui] Plane: Plane [name=plane3, companyName=Company, transportSize=84, tt=Civile, isFree= false] Pilot: []

Table 3. *Cont.*

S_i	$Inputs_i$	$Outputs_i$
S9	ID: e19d46eb-a534-4891-a51a-7e5fcc4f9f3f Quantity: 0 Transport Type: Merchandise Destination Town: [Ville [name=, pays=, location=Local], Ville [name=, pays=, location=Europe]] Planes: Heavy Planes: [Plane [name=plane, companyName=company, transportSize=1500, tt=Merchandise, isFree=true]] Light Planes: [Plane [name=plane, companyName=company, transportSize=999, tt=Merchandise, isFree=true]] Civil Planes: [Plane [name=plane, companyName=company, transportSize=100, tt=Civil, isFree=true]] Pilots: Pilots Local: [Pilot [name=pilot, companyName=company, workingDest=Local, isFree=false]] Pilots Europe: [Pilot [name=pilot, companyName=company, workingDest=Europe, isFree=true]] Fitness Value: 1.75 Percent:87.5%	ID: e19d46eb-a534-4891a51a-7e5fcc4f9f3f Trace: [MainGuiToPilot, APilotToAPL, APLToAPilotF, APilotToAPEU, APEUToAPilotI, APilotToMainGui] Plane: null Pilot: [null, Pilot [name=pilot, companyName=company, workingDest=Europe, isFree=false]]
S14	ID: 3299f371-b01d-4edb-a439-d141936dac13 Quantity: 0 Transport Type: Civile Destination Town: [Ville [name=, pays=, location=Europe]] Planes: Heavy Planes: [Plane [name=plane, companyName=company, transportSize=1500, tt=Merchandise, isFree=true]] Light Planes: [Plane [name=plane, companyName=company, transportSize=999, tt=Merchandise, isFree=true]] Civil Planes: [Plane [name=plane, companyName=company, transportSize=100, tt=Civil, isFree=true]] Pilots: Pilots Local: [Pilot [name=pilot, companyName=company, workingDest=Local, isFree=true]] Pilots Europe: [Pilot [name=pilot, companyName=company, workingDest=Europe, isFree=true], Pilot [name=pilot, companyName=company3, workingDest=Europe, isFree=true], Pilot [name=pilot1, companyName=company1, workingDest=Europe, isFree=true]] Fitness Value: 2 Percent:100%	ID: 3299f371-b01d-4edb-a439-d141936dac13 Trace: [MainGuiToPilot, APilotToAPEU, APEUToAPilotI, APilotToMainGui] Plane: null Pilot: Pilot [name=pilot, companyName=company, workingDest=Europe, isFree=false]

4.3. Generation of the Expected Outputs

This phase consists of finding the expected outputs $Outputs_i$ for the inputs $Inputs_i$ generated for each scenario S_i. For this, it suffices to apply to the inputs $Inputs_i$ generated for each scenario S_i and the set of activities $Activitys_i$ that will be triggered by the interactions of this latter. These activities are stored in the nodes of the graph G. Table 3 represents a subset of test case inputs with their expected outputs (one test case for some scenarios), obtained as a result of applying our approach for testing the airport management system.

4.4. Detection of Errors in the System

The last phase of the proposed approach consists of running the original system with the inputs $Intputs_i$ presented in Table 3 and comparing the results obtained with those expected $Outputs_i$ also presented in Table 3. As an example, Figures 7 and 8, respectively, represent the end of the test operation for the two scenarios S_1 and S_2, wherein in the second scenario, an error is detected.

The detection of this error was carried out following the comparison between the expected outputs for the system's execution with the test case inputs generated for scenario S_2 and the outputs of the system's execution with the same inputs. Figure 9 shows that the execution outputs are incompatible with the expected outputs. In fact, in the execution outputs, the system's agents have executed two more interactions. The execution of these additional interactions is not caused by the execution of other scenarios in parallel with scenario S_2 which is under test. Indeed, despite the possibility of the execution of scenario S_2, in parallel with other scenarios, these latter were not executed. This is guaranteed by

the test case inputs generated for each scenario in the first phase of our approach. Indeed, the test case inputs generated for scenario S_2 only permit the execution of the interactions and activities of this scenario, not those of any other scenarios. Consequently, during the execution of the system with these inputs, to test scenario S_2, the agents find themselves forced to only execute scenario S_2, which is under test. In this way, it will be possible to ensure that the error detected in the latter is not associated with another scenario operating in parallel.

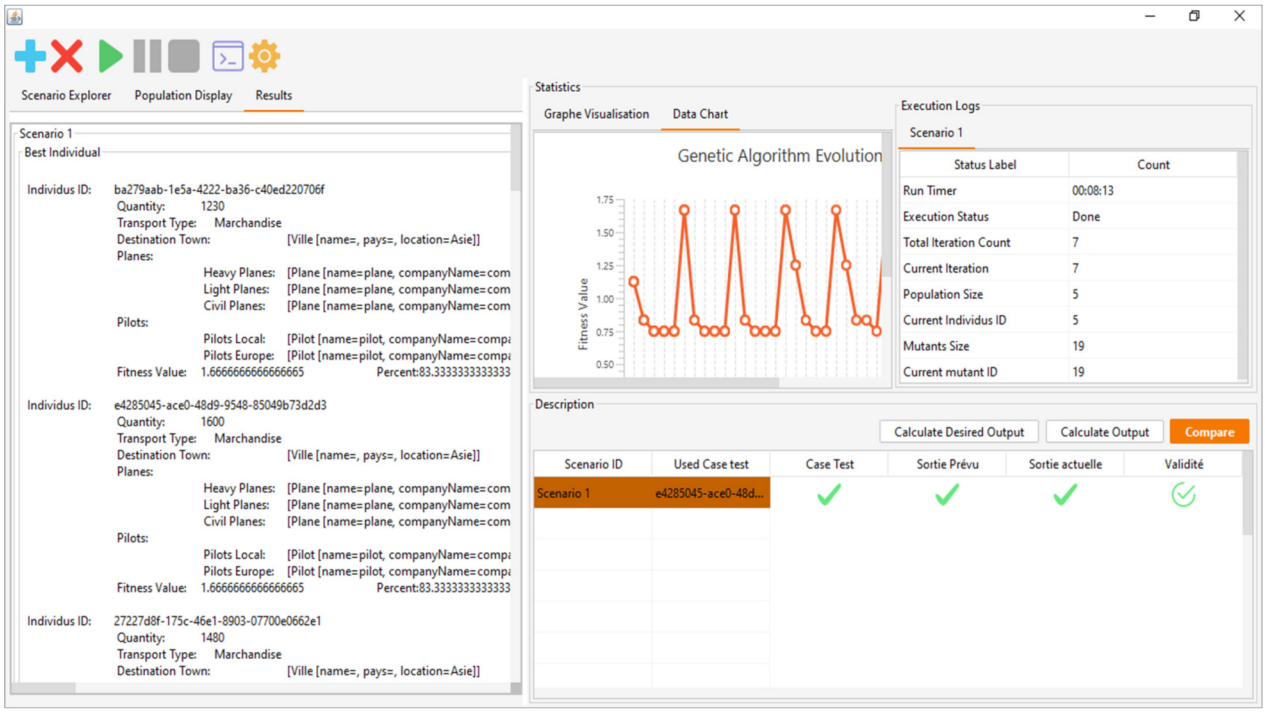


Figure 7. End of the test of scenario 1 (no errors).

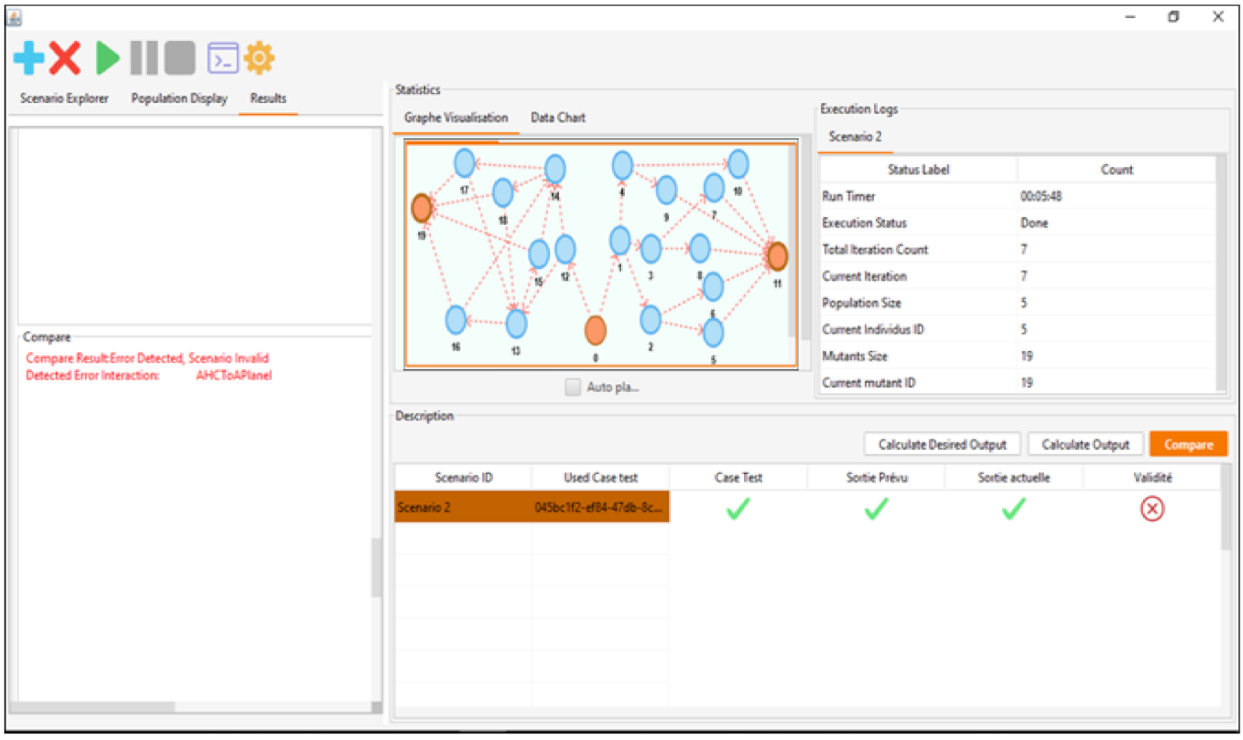


Figure 8. End of the test of scenario 2 (detection of an error).

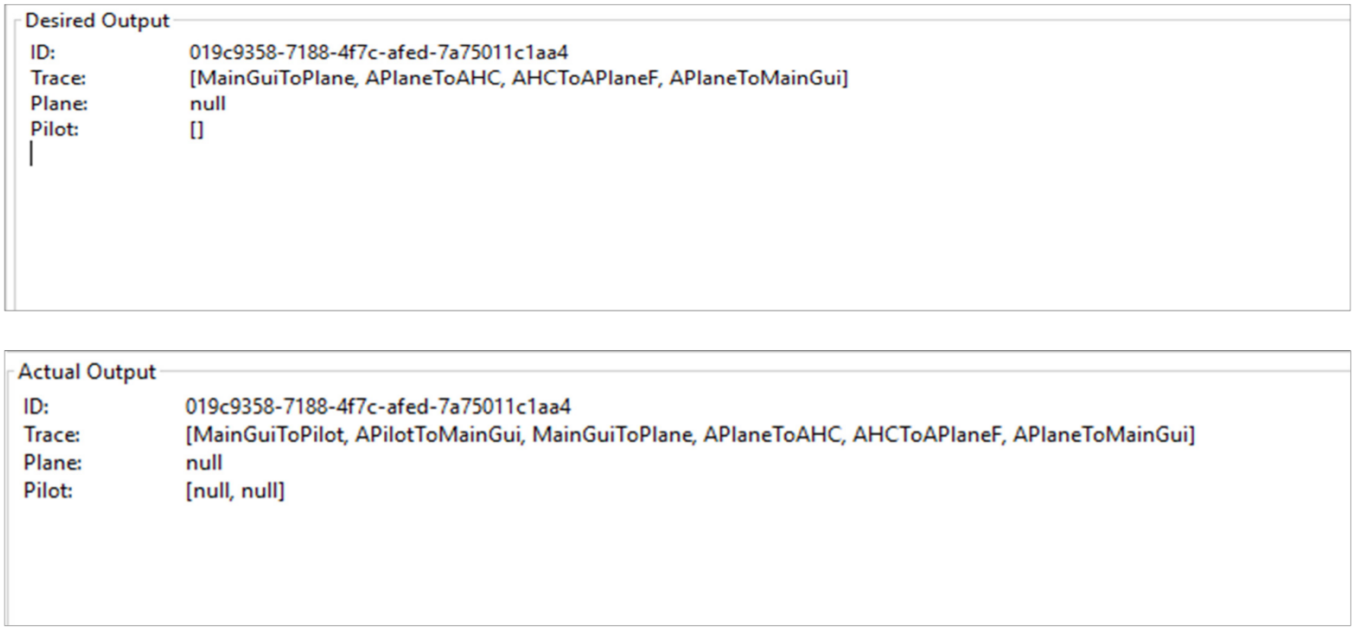


Figure 9. Comparison between expected outputs and execution outputs.

4.5. Discussion and Limitations

Applying our testing approach to the selected case study allowed us to demonstrate the necessity of testing behavioural scenarios individually. In fact, this has enabled us to identify, among the scenarios running in parallel, the scenario that caused the observed error in the event of error detection and, therefore, to facilitate the correction of the detected errors.

This example has also allowed us to show that our approach has several advantages, namely:

- The use of the mutation analysis technique has allowed us to facilitate and optimise the generation of test case inputs that may cover each scenario under test individually. As a result, this enables us to exceed the traditional test methodologies, which include introducing plugs and employing sophisticated and complex algorithms to generate test case inputs.
- The use of parallel genetic algorithms allows us to speed up the time required for the generation of test case inputs. Indeed, compared to the usage of non-parallel genetic algorithms, our approach ensures higher speed because each algorithm Ai is responsible for discovering the test case inputs associated with a single scenario S_i. Therefore, the size of the system under test and the number of behaviour scenarios it contains do not affect the calculation of the time required to generate test case inputs.
- The use of the sequence and activity diagram has allowed us to automate the generation of test case outputs. As a result, the method may be (i) very useful for software that uses models in the analysis and design phases and (ii) easily adapted to systems providing the execution of several behavioural scenarios simultaneously.

As for limitations, our approach cannot be applied to open multi-agent systems where new scenarios can be introduced following the participation of new agents in the system [30,31]. In fact, test case inputs generated using our approach to ensure the individual coverage of behavioural scenarios available when testing the system may not be relevant with new scenarios introduced as a result of the participation of new agents. Moreover, the generation of test case outputs will be difficult because it will be based on meta-models that describe the behaviour that will be acquired by new agents participating in the system.

5. Conclusions

The testing activity represents an important task in the software quality assurance process. Although it is laborious and costly, the importance of this activity in terms of the reliability and quality of the software as well as economically encourages us to give particular interest to the development of tools allowing the automation of its different phases. Despite the rapid evolution of MAS, the testing of these systems is still a key open area. Despite the fact that only a few proposals for MAS testing are available in the literature, they could contribute significantly to the progress of the field of MAS testing. However, most of these proposals are related to unit-level testing and agent-level tests. This leaves several issues related to system-level tests unresolved, such as the problem of the possibility of executing several behavioural scenarios at the same time by the agents of the system under test. In this work, we presented a new test case generation approach capable of covering behavioural scenarios individually in a multi-agent system. The objective is to know exactly, in the case of detection of an error, the scenario that caused the detected error, among the scenarios running in parallel. The proposed approach, supported by the tool that we have developed, has been validated in a concrete case study: Airport management system. According to the obtained results, it would be interesting to incorporate our tool into agent development platforms as a separate library to support the process of testing behavioural scenarios in MAS. In prospect, we plan in the short and medium terms to adapt our approach with open multi-agent systems.

Author Contributions: Conceptualization, N.E.H.D.; methodology, N.E.H.D.; software, A.H.B. and N.E.H.D.; validation, N.E.H.D., A.H.B. and Z.T.; formal analysis, N.E.H.D.; investigation, N.E.H.D.; resources, N.E.H.D. and A.H.B.; data curation, N.E.H.D. and A.H.B.; writing—original draft preparation, N.E.H.D.; writing—review and editing, N.E.H.D. and Z.T.; visualization, N.E.H.D.; supervision, N.E.H.D.; project administration, N.E.H.D.; funding acquisition, N.E.H.D. and Z.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kiran, A.; Butt, W.H.; Anwar, M.W.; Azam, F.; Maqbool, B. A Comprehensive Investigation of Modern Test Suite Optimization Trends, Tools and Techniques. *IEEE Access* **2019**, *7*, 89093–89117. [CrossRef]
2. Zardari, S.; Alam, S.; Al Salem, H.A.; Al Reshan, M.S.; Shaikh, A.; Malik, A.F.K.; Rehman, M.M.U.; Mouratidis, H. A Comprehensive Bibliometric Assessment on Software Testing (2016–2021). *Electronics* **2022**, *11*, 1984. [CrossRef]
3. Nguyen, C.D.; Perini, A.; Bernon, C.; Pavón, J.; Thangarajah, J. Testing in multi-agent systems. In Proceedings of the International Workshop on Agent-Oriented Software Engineering, Budapest, Hungary, 11–12 May 2009; pp. 180–190.
4. Zhang, Z.; Thangarajah, J.; Padgham, L. Automated unit testing for agent systems. In Proceedings of the 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Barcelona, Spain, 23–25 July 2007; pp. 10–18.
5. Zhang, Z.; Thangarajah, J.; Padgham, L. Automated unit testing intelligent agents in pdt. In Proceedings of the AAMAS, Estoril, Portugal, 12–16 May 2008; pp. 1673–1674.
6. Zhang, Z.; Thangarajah, J.; Padgham, L. Model based testing for agent systems. In Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, Budapest, Hungary, 10–15 May 2009; pp. 1333–1334.
7. Ekinci, E.E.; Tiryaki, A.M.; Cetin, O.; Dikenelli, O. Goal-Oriented Agent Testing RevisitedIn Proceedings of the 9th Int. Workshop on Agent-Oriented Software Engineering, Estoril, Portugal, 12–13 May 2008; pp. 85–96.
8. Nguyen, C.D.; Perini, A.; Tonella, P. Goal-oriented testing for MASs. *Int. J. Agent-Oriented Softw. Eng.* **2010**, *4*, 79–109. [CrossRef]
9. Padgham, L.; Zhang, Z.; Thangarajah, J.; Miller, T. Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1230–1244. [CrossRef]
10. Coelho, R.; Kulesza, U.; Von Staa, A.; Lucena, C. Unit testing in multi-agent systems using mock agents and aspects. In *International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*; ACM Press: New York, NY, USA, 2006; pp. 83–90.
11. Lam, D.N.; Barber, K.S. Debugging agent behaviour in an implemented agent system. In Proceedings of the Second International Workshop ProMAS, New York, NY, USA, 20 July 2004; pp. 104–125.
12. Nguyen, C.D.; Miles, S.; Perini, A.; Tonella, P.; Harman, M.; Luck, M. Evolutionary testing of autonomous software agents. *Auton. Agents Multi-Agent Syst.* **2011**, *25*, 260–283. [CrossRef]
13. Nguyen, C.D.; Perini, A.; Tonella, P. Ontology-based Test Generation for Multi Agent Systems. In Proceedings of the 7th International Conference on Autonomous Agents and Multi Agent Systems, Estoril, Portugal, 12–16 May 2008; pp. 1315–1320.

14. Núñez, M.; Rodríguez, I.; Rubio, F. Specification and testing of autonomous agents in e-commerce systems. *Softw. Test. Verif. Reliab.* **2005**, *15*, 211–233. [CrossRef]
15. Clark, A.G.; Walkinshaw, N.; Hierons, R.M. Test case generation for agent-based models: A systematic literature review. *Inf. Softw. Technol.* **2021**, *135*, 106567. [CrossRef]
16. De Wolf, T.; Samaey, G.; Holvoet, T. Engineering self-organising emergent systems with simulation-based scientific analysis. In Proceedings of the Third International Workshop on Engineering Self-Organising Application, Utrecht, The Netherlands, 25 July 2005; pp. 146–160.
17. Dehimi, N.E.H.; Mokhati, F.; Badri, M. Testing HMAS-based applications: An ASPECS-based approach. *Eng. Appl. Artif. Intell.* **2015**, *46*, 232–257. [CrossRef]
18. Dehimi, N.E.H.; Mokhati, F. A Novel Test Case Generation Approach based on AUML sequence diagram. In Proceedings of the International Conference on Networking and Advanced Systems (ICNAS), Annaba, Algeria, 26–27 June 2019. [CrossRef]
19. Thangarajah, J.; Harland, J.; Morley, D.N.; Yorke-Smith, N. Towards quantifying the completeness of BDI goals. In Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'14), Paris, France, 5–9 May 2014; pp. 1369–1370.
20. Gonçalves, E.M.N.; Machado, R.A.; Rodrigues, B.C.; Adamatti, D. CPN4M: Testing Multi-Agent Systems under Organizational Model Moise+ Using Colored Petri Nets. *Appl. Sci.* **2022**, *12*, 5857. [CrossRef]
21. Papadakis, M.; Kintis, M.; Zhang, J.; Jia, Y.; Le Traon, Y.; Harman, M. Chapter Six—Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.* **2019**, *112*, 275–378. [CrossRef]
22. Alexander, R.; Bieman, M.; Sudipto, G.; Bixia, J. Mutation of Java Objects. In Proceedings of the 13th International Symposium on Software Reliability Engineering, Annapolis, MD, USA, 12–15 November 2002.
23. Chevalley, P. Applying mutation analysis for object-oriented programs using a reflective approach. In Proceedings of the Eighth Asia-Pacific Software Engineering Conference, Macao, China, 4–7 December 2001.
24. Ghosh, S.; Mathur, A. Interface Mutation to assess the adequacy of tests for components and systems. In Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems—TOOLS, Santa Barbara, CA, USA, 4 August 2000.
25. Savarimuthu, S.; Winikoff, M. Mutation Operators for the Goal Agent Language. In Proceedings of the International Workshop on Engineering Multi-Agent Systems, St. Paul, MN, USA, 6–7 May 2013.
26. Huang, Z.; Alexander, R.; Clark, J. Mutation Testing for Jason Agents. In Proceedings of the International Workshop on Engineering Multi-Agent Systems, Second International Workshop, EMAS 2014, Paris, France, 5–6 May 2014.
27. Abu Bakar, N.; Selamat, A. Agent systems verification: Systematic literature review and mapping. *Appl. Intell.* **2018**, *48*, 1251–1274. [CrossRef]
28. Barnier, C.; Aktouf, O.-E.-K.; Mercier, A.; Jamont, J.P. Toward an Embedded Multi-agent System Methodology and Positioning on Testing. In Proceedings of the 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Toulouse, France, 23–26 October 2017; pp. 239–244.
29. Winikoff, M. BDI agent testability revisited. *Auton. Agents Multi-Agent Syst.* **2017**, *31*, 1094–1132. [CrossRef]
30. Hendrickx, J.M.; Martin, S. Open multi-agent systems: Gossiping with deterministic arrivals and departures. In Proceedings of the 54th Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, USA, 27–30 September 2016.
31. Hendrickx, J.M.; Martin, S. Open multi-agent systems: Gossiping with random arrivals and departures. In Proceedings of the 2017 IEEE 56th Annual Conference on Decision and Control (CDC), Melbourne, VIC, Australia, 12–15 December 2017.