PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 16/10

A Multi-Agent System Framework to Assure the Reliability of Self-Adapted Behaviors

Andrew Diniz da Costa
Viviane Torres da Silva
Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 16/10 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena November, 2010

A Multi-Agent System Framework to Assure the Reliability of Self-Adapted Behaviors

Andrew Diniz da Costa, Viviane Torres da Silva[1],
Carlos José Pereira de Lucena

[1]Departamento de Informática – Universidade Federal Fluminense (UFF)

acosta@inf.puc-rio.br, viviane.silva@ic.uff.br, lucena@inf.puc-rio.br

Abstract. One of the most important issues in self-adaptive systems is to assure the reliability of self-adapted behaviors. However, few works provide solution to deal with such concern. The majority considers the execution of test cases to validate the self-adapted behavior. Based on this idea, the paper proposes a framework that helps on building self-adaptive and self-testable multi-agent system This framework allows the creation of different self-adaptation processes (control-loops) composed of a set of activities (e.g. collect, analyze, plan, execute, etc). In order to help on testing and validating the adapted behavior, the framework provides two main activities (*test* and *validate*) and a set of elements that help on representing the self-test concept in different control-loops. The applicability of the framework is demonstrated by an industrial system, developed to a petroleum company, responsible for identifying routes that attend products derived from the petroleum (e.g. kerosene, gasoline, etc) in different places in Brazil.

Keywords: Design, Reliability, Experimentation, Security and Verification.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Table of Contents

1 Introduction

The development of complex systems, which are able to self-adapt theirs behaviors when necessary, is becoming extremely common. In this context, the multi-agent system paradigm (Wooldridge and Jennings, 1998) has been used especially when distributed, autonomous and pro-active entities are represented.

In order to deal with such tendency, several approaches have been proposed to help on building self-adapted systems. However, few works, such as (Denaro et al., 2007), (Costa et al, 2010), (Stevens et al, 2007) and (Wen et al, 2005) provide solution to deal with one of the most important issues in self-adaptive systems that is to assure the reliability of the self-adapted behavior. The main idea of such approaches is to execute tests before effectively adapt the behavior. Thus, it is possible to evaluate if a suggested adaptation is adequate or not.

Although it is possible to define tests to be executed to test the adapted behavior, such approaches provide little support to their definition. After analyzing several approaches published in the literature, we have figured out that a set of important concerns are not represented by them:

- Relating artifacts to be tested to their test cases. It is not only important to define the tests themselves but also to relate them to the artifacts (or behavior) that they are able to test;
- Defining test conditions of a test case, i.e., to define different sets of input data and output assertions to a specific test. By doing this, the use of a given test case become more flexible since it can be executed in different ways for different purpose;
- Helping on the creation of tests able to read stored data, i.e. data stored in databases, files, etc. This is an important concern because tests that use persisted data are common in complex systems;
- Defining the order of the execution of the tests used for validating a given artifact. This concern is important because the execution of one test may depend on the results of another test;
- Defining different log formats to store the results of the performed tests. Thus, each system can define different ways of handling them;
- Allowing the use of the self-test activity in other self-adaptation processes, i.e. such activity should not be tied to only one process.

The goal of the paper is to present the Self-Test (ST) Framework that aims to allow the creation of multi-agent systems that perform self-test before executing the self-adapted behavior. ST framework enables the construction of self-adaptive agents that can execute different self-adaptation processes (control-loops) composed of activities that can perform the collection of data, analysis, decisions, etc. The main features of the framework are: (i) to provide two activities (*test* and *validate*) that represent the self-test concept and that can be used in different self-adaptation processes; (ii) to allow the creation of different control-loops; (iii) to provide a default control-loop that uses the *test* and *validate* activities; and (iv) to define, by using XML files, the test cases that will be used to validate the artifacts.

The paper is organized as follows. In Section 2 we present related works. Section 3 details the ST Framework. In Section 4 a case study is described and finally, in Section 5, conclusions and some future works are presented.

2 Related Work

The main focus of the work presented in (King et al., 2007) is to provide a self- test framework that should be used at microprocessors. The framework consists of two main steps: (i) self-test of on-chip processor core(s), and (ii) test and response analysis of other on-chip components, using the tested processor core(s) as the pattern generator and response analyzer. Since the components of the framework are tied to concepts related to microprocessors, it is not possible to use it in different domains, such as the approach being proposed in this paper.

In (Denaro et al., 2007) the authors presents a self-adaptive approach that uses a control loop structured in the following way: monitoring mechanisms, diagnosis mechanisms, and adaptation strategies. The self-adaptive approach implemented consists of a pre-processing and a generation step. The pre-processing test is composed of: the identification of possible integration problems, the generation of test cases for revealing integration problems, and the definition of suitable recovery actions. Even considering such structure, the approach does not consider important steps defined in the proposal being presented in this paper, such as the possibility to define input data and output assertions to be used in different tests and to define the order of execution of the responsible tests for validating the artifacts (e.g. behavior, web-service, etc). In addition and different to our approach, the self-test approach presented in (Denaro et al., 2007) proposes a fix process. In our approach different control loops can be created.

The work in (Stevens et al, 2007) proposes a framework for testing self-adaptive systems. Such work introduces the concept of an autonomic container, which is a data structure that has self-managing capabilities, and also has the implicit ability to self-test. Such approach uses a strategy that tests copies of managed resources while the actual managed resource is being used by a system, a technique known as *replication with validation*. However, it does not allow the use of the self-test activities in other architectures because the framework is tied to the IBM's layered architecture for autonomic computing (AC) systems (IBM, 2003). Therefore, it is not possible to use the self-test activities in different self-adaptation processes. Besides, the framework does not help the creation of test cases that recovery data from different bases what is frequently used when testing and validating complex self-adapt systems.

The approach in (Costa et al., 2010) proposes the *Java self-Adaptive Agent Framework* with self-Test (JAAF+T) that extends the JADE framework (Bellifemine et al., 2007), which already gives support to the implementation of autonomous and pro-active agents. JAAF+T allows the creation of self-adaptive agents that can execute different self-adaptation processes and activities that compose such processes. However, the framework does not provide support to the creation of tests that need to recovery data stored in different bases. Although JAAF+T provides files that represent test cases and data that can be used in self-adaptation processes, the framework does not provide an intuitive way to relate test cases with artifacts that must be tested, and to define test conditions to a same test case. Such disadvantages are treated by the approach being proposed in this paper.

3 Self-Test Framework

As previously mentioned one of the most important issues in self-adaptive systems is to assure the reliability of the self-adapted behavior. However, few works, such as (Denaro et al., 2007), (Costa et al, 2010), (Stevens et al, 2007) and (Wen et al, 2005) provide solution to deal with such issue.

The approaches used by (Costa et al, 2010) and (Stevens et al, 2007) execute tests in the proposed adaptation in order to evaluate its reliability. However, such approaches have some drawbacks. For instance, they do not help the designers on the creation of tests able to read stored data, i.e. data stored in databases, files, etc. Based on this idea, the Self-Test (ST) framework allows the creation of self-tests that can use data stored in different bases.

Initially, this section presents the main idea of the ST framework (section 3.1). Next, a set of XML files that helps on defining the testes is presented (section 3.2). After, the hot-spots and frozen-spots of the framework are described (section 3.3). And finally, a step by step description of how to use the ST framework is presented.

3.1 Main Idea

Before adapting its behavior, the agent should execute a set of tests in order to validate its adaptation. Aiming to contemplate this idea, we extended the JADE framework (Bellifemine et al., 2007), a FIPA compliant framework developed in Java to implement multi-agent systems (MASs), in order to represent the following concepts: (i) self-adaptation plans (or control-loops) that makes JADE agents able to perform self-tests; (ii) activities that are the steps of such plans, in particular two activities called Test and Validate that can be re-used in different control-loops; and (iii) a set of XML files that allows representing the tests to be executed.

To facilitate the work of developers, the ST framework provides a control-loop based on the default self-adaptation process proposed in (IBM, 2003). Such control-loop is composed of six activities (*Collect*, *Analyze*, *Decision*, *Test*, *Validate* and *Effect*) as illustrated in Figure 1.

Collect activity is responsible for recovering data that help to identify the problem and on choosing the action to be adapted. Besides, such activity formats the data recovered according to a structure readable by other activities. Next, the *Analyze* activity performs a diagnosis based on the recovered data in order to verify if a self-adaptation will be necessary.

After meeting the diagnosis, the *Decision* activity looks for an action (or behavior, web-service, etc.) that may help the agent on achieving its goal. However, before confirming the choice, the action is sent to the *Test* activity, which is responsible for executing tests on such action. When the execution is finished, the results of the tests are provided to the *Validate* activity that is responsible to analyze the results of the tests, and decide if the action is a valid one or if it is necessary to choose another one. If the action is approved, the *Effect* activity is executed in order to configure the agent with the chosen action. On the other hand, the *Decision* activity is executed again to choose another action.

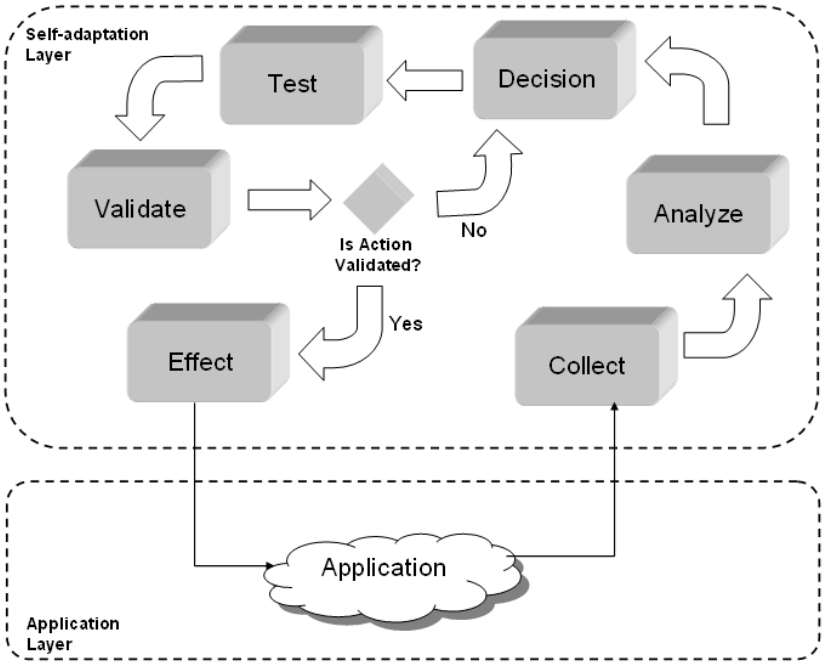

Figure 1. Control-Loop provided by the Self-T framework

Figure 2 illustrates a class diagram of the framework with its main classes. By using the *ControlLoop* class it is possible to define different self-adaptation plans. Such class extends the *FSMBehaviour* that allows to define the new activities (by extending the *Activity* class) based on the conception of finite state machines. The ST framework already provides the control-loop illustrated in Figure 1 and implemented by *CLWithSelfTest* class that has the six activities represented by *Collect*, *Analyze*, *Decision*, *Test*, *Validation* and *Effect* classes. Besides, in order to represent data that can be handled by different activities, the *HandledData* class was defined.

Test and *Validate* classes represent the main activities that allow applying the self-test concept in a control-loop. The first activity executes a set of tests in a given artifact (such as a behavior or web-service) by using the *StartTests* class, while the second activity evaluates the results of the executed tests by reading a set of logs from the *ReadingLog* class in order to evaluate if a tested action really can be executed.

The description of the test cases that will be executed, the control-flow (or running cycle) of the tests to be executed, and the data used as input and to be confirmed as output are defined in the following XML files, TL, CFL and DL, respectively. Details of such files and of the classes used by the *Test* activity are presented in the next subsections.

3.2 XML Files

As mentioned previously, ST framework provides three XML files: TL, CFL and DL. TL (Test Language) file describes the test cases that can be executed in the system. CFL (Control Flow Language) file describes the control-flow for the execution of the tests related to each artifact that must be tested, and DL (Data Language) file represents the input data and output assertions of each test case.

In order to use the ST framework to perform self-test, it is necessary to provide such files before the execution of the framework. Aiming to clarify the use of such files, examples illustrating the structure of each one are presented in the following.

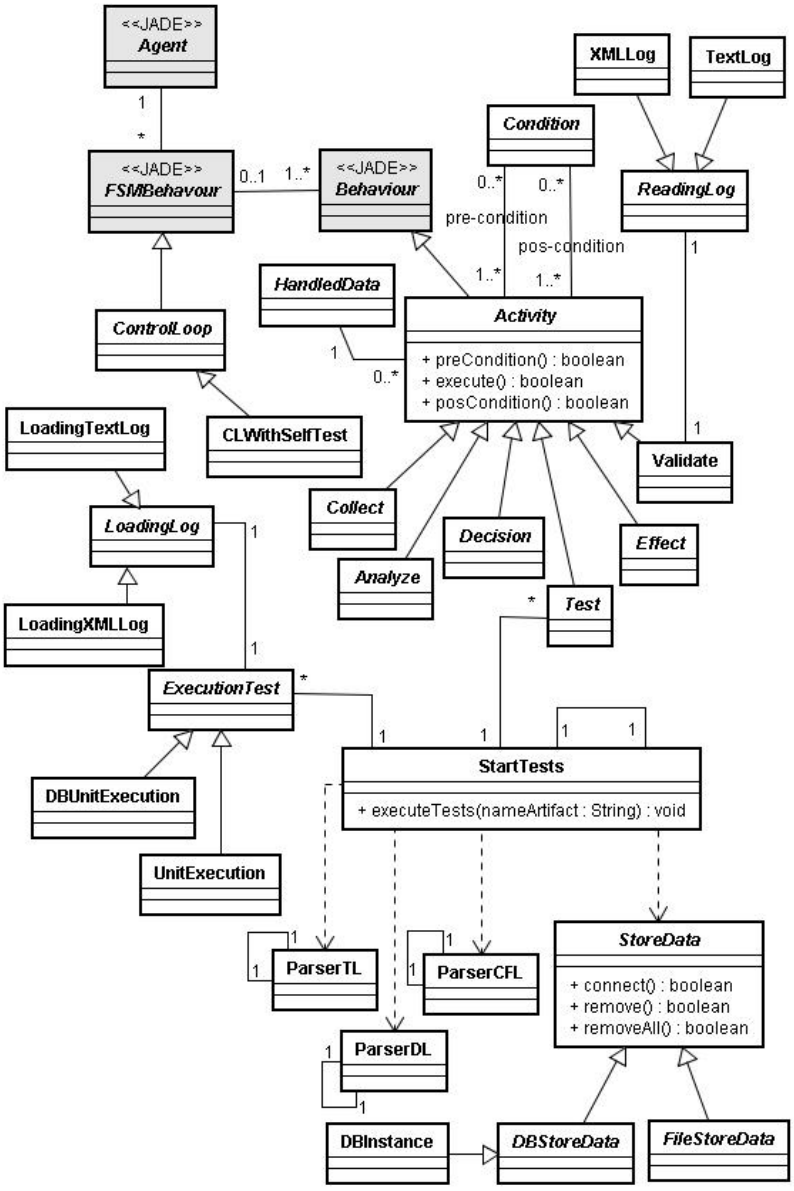

Figure 2. Main Class Diagram to represent self-test

Figure 3 presents an example of the TL file. It describes two sets of test cases, each one described by using the *setTestCase* element. Each test case set has attributes that can be used to identify the set, such as the version of the system to be tested, the type of tests to be executed (e.g. unit, database, integration or functional), the name of the set (that is a mandatory information), and the context related to the tests (e.g. e-commerce, traffic-control, oil's system, etc.). Each test case set can be composed of several test cases. A test case is identified by a name, the type of the test to be executed (e.g. JUnit or DBUnit) and the path of the test case that will be executed (*classname* element). In addition, it is also possible to indicate the priority of each test by taking into account

only the tests defined in the same set. The above mentioned data, except for the priority of the test cases, are mandatory. *ParserTL* class (see Figure 2) is responsible to read TL files.

```
<testcases>
    <setTestCase version="1.0" type="Unit"
        name="settestcase1" context="oil-system">
        <testcase name="testCase1" type="JUnit">
            <classname>
                testcases.TestCase1
            </classname>
            <priority>1</priority>
        </testcase>
        <testcase name="testCase2" type="DBUnit">
            <classname>
                testcases.TestCase2
            </classname>
            <priority>2</priority>
        </testcase>
    </setTestCase>
    <setTestCase version="1.0" type="Functional"
        name="settestcase2" context="oil-system">
        ...
    </setTestCase>
    ...
</testcases>
```

Figure 3. An example of the TL file

In Figure 4 part of a CF file is illustrated. It presents two artifacts that should be tested. For each artifact it is necessary to inform: (i) its type (*type* attribute), for instance *behaviors*; (ii) an identifier that informs where such artifact is located (*nameArtifact* attribute); (iii) the path of the file that will store the results of the tests to be executed (*log* attribute); and (iv) the tests to be used to validate such artifact (*test* element).

Test element has two attributes: *type* and *name*. The first attribute informs if the element is related to a test case (*testcase*) or to a set of test cases (*setTestCase*). The second attribute informs the name of the test case or the name of the set defined in the TL file. In our example, the first artifact is related to the *settestcase1* defined in the TL file presented in Figure 3. The second artifact is related to a test case called *tetsCase3*, which must be defined in another TL file.

The framework considers that the order of the tests described in the CFL file is the execution order. However, if a *test* element is related to a set of tests, the order will be based on the priority defined for each test case represented in the TL file (see *priority* element in Figure 3).

The DL file is responsible for representing the input and output assertions used for each test case. Figure 5 illustrates part of a DL file with its elements and attributes. The *data* element defines the data to be used as input and output in each test case. Therefore, one test case must be indicated for each *data* element by using the attribute called *testCaseRelated*. In addition, each test case has a *condition* element that defines the sets of input data and output assertion to be used when testing and validating the artifact.

The input and output data represented in the DL file can be used to store the data that will be directly used by the test case or to store the data that will be saved in a database or file used by the test case to read the desired information. Note that the output assertions are data that are used to verify if the test obtained the expected result.

```
<control-flow>
    <artifact
        type="behaviour"
        nameArtifact="behaviours.Behaviour1"
        log="C:/eclipseEuropa/eclipse/teste.xml">
        <test type="setTestCase" name="settestcase1" />
    </artifact>
    <artifact
        type="behaviour"
        nameArtifact="behaviours.Behaviour2"
        log="C:/eclipseEuropa/eclipse/teste3.txt">
        <test type="testcase" name="testCase3" />
    </artifact>
</control-flow>
```

Figure 4. An example of the CF file

```
<dl>
    <data
        name_context="context1"
        testCaseRelated="testCase1">
        <condition name="condition1"
            type_data="database" save="false"
            deleteBeforeTest="true">
            <setInputs destiny="table1">
                <input name="column1" value="10"/>
                <input name="column2" value="100"/>
            </setInputs>
            <setInputs destiny="table2">
                <input name="column1" value="15"/>
                <input name="column2" value="200"/>
            </setInputs>
            <setOutputs destiny="table3" store="false">
                <output name="column1" value="1000"/>
            </setOutputs>
        </condition>
        ...
    </data>
    ...
</dl>
```

Figure 5. An example of the DL file

In order to define such difference, the *save* attribute (save=true) must be used to inform if the data must be saved in a database or file. If the data should be persisted, the developer must inform in the *type_data* attribute the type of the base where such data will be stored. Nowadays, the framework allows storing data in two different base types: database or a text file.

In the example of a DL file illustrated in Figure 5 the input and output data related to *testCase1* (test cases identified in TL file) are stores in a database. For each input and output element, the name of the table that stores the data is informed in the *destiny* attribute and the column name where the data is stored is informed in the *name* attribute. This attribute must have the same column name of the column represented in the provided table.

Note that for each input or output element it is possible to define if such information will be stored or not. In order to do so the *store* attribute associated with the setInputs and setOutputs elements should be used. In Figure 5 the output data of the *testCase1* should not be stored. Considering this case, the tests will work with the transaction concept, i.e., commits are not performed in the database used.

Data about connections with a database or a file (e.g. url, login, password) should be informed in a properties file provided by the framework. However, if the developer wants to define other configurations, the *DBStoreData* and *FileStoreData* classes (see Figure 2) should be extended, respectively.

It is also possible to inform by using the attribute *deleteBeforeTest* of the DL file if the data stored in the tables informed by using the attribute *destiny* should be deleted or not before executing the test.

Figure 6 illustrates the dependences between the XML files provided by the ST framework. Note that CFL and DL files depend on the TL file, because they reference test cases defined in such file.

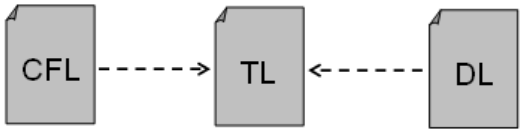

Figure 6. Relation between CFL, TL and DL.

3.3 Hot-Spots and Frozen-Spots

This section describes in detail the hot-spots (flexible points) and frozen-spots (fixed points) (Fayad et al., 1999) of the ST framework. The hot-spots defined by the framework are:

- The creation of control-loops: by extending the *ControlLoop* class, different self-adaptation plans can be created representing different control loops. The default control-loop provided by the framework is the control-loop for self-adaptive systems being proposed in this paper and illustrated in Figure 1.
- The creation of activities: by extending the *Activity* class, new activities for different control-loops can be created. Such class provides the following t-hree methods that should be implemented by its sub-classes:: (i) *preCondition* that should be called before executing the activity in order to check the pre-condition for such execution; (ii) *execute* that represents the main execution of the related activity; and (iii) *posCondition* that should be called after executing the activity in order to check the post-condition of such execution.
- The definition of new conditions: by using the *Condition* class it is possible to define pre- and pos-conditions of activities defined to a given control-loop.

- The definition of different types of tests: In order to allow the execution of different types of tests (e.g. unit, functional, etc) the *ExecutionTest* class was defined. The framework already provides support to the execution of unit tests by using the JUnit (JUnit, 2010) and DBUnit API (DBUnit, 2010). In order to implement other types the *ExecutionTest* class must be extended.
- The reading of different logs: By extending the *ReadingLog* class it is responsible to read different types of logs.
- The writing in different logs: By extending the LoadingLog class it is possible to load the result of tests saved in different log's formats.
- The definition of different ways of persisting the data to be used in the tests: By extending the *StoreData* class different ways of persisting the data to be used in tests can be defined. The framework already provides two options: the use of a database (*DBStoreData* class) or the use of a file (*FileStoreData* class).

The frozen-spots of the framework are:

- The control-loop provided by the framework: ST framework provides the control-loop presented in Figure 1 (*CLWithSelfTest* class), composed of six activities (*Collect*, *Analyze*, *Decision*, *Test*, *Validate* and *Effect*).
- The support for reading two different types of logs: The framework provides support for reading two types of logs: XML (*XMLLog* class) and Text (*TextLog* class).
- The parsing of TL file: ST framework provides support for the reading of the TL file by using the *ParserTL* class that was implemented following the Singleton pattern (Gamma et al., 1995).
- The parsing of CFL file: In order to read the data informed in the CFL file, the framework provides the *ParserCFL* class, also implemented following the Singleton pattern.
- The parsing of DL file: As well as the TL and CFL files, the DL file also is a singleton (*ParserDL*) that allows the reading of data in the DL format.
- The support for easily execute the testes: In order to execute the desired tests, the framework provides the *StartTests* class. Such class applies the *Façade* pattern (Gamma et al., 1995) in order to simplify the execution of the tests. Thus, to execute test cases the *executeTests* is the unique method that must be called.

3.4 Using ST Framework

In order to implement a self-adaptive agent that performs self-test, the main steps should be followed:

1) Define in TL the test cases that should be executed to evaluate the self-adaptations.
2) Implement the different types of test executions (e.g. functional, integration) by extending the *ExecutionTest* class. As mentioned previously, the framework al-

ready gives support to the execution of unit tests created by using the JUnit and DBUnit APIs.

3) Define in DL, the input and output data used in each test case.
4) Implement different treating of access of data stored in bases by using the *StoreData* class.
5) Define by using *LoadingLog* and *ReadingLog* classes how the results of the executed tests will be written and read, respectively.
6) Define by using CFL the control-flow of tests.
7) Use the default control-loop provided by the framework or create a new one by using the *ControlLoop* class. If the developer wants to create new activities, he should extend the *Activity* class.
8) Create a software agent by extending the *Agent* class, provided by the JADE framework, and associate it with the control-loop defined in the previous step.

4 Case Study: Petroleum Control System

The Software Engineering Laboratory of the Pontifical Catholic University of Rio de Janeiro has extensively worked on developing legacy systems and on coordinating and carrying out tests of such systems developed to a Brazilian petroleum company.

The application that we will explore in this paper is responsible to: (i) register routes (i.e. paths) based on ducts and ships that could be used to transport the derived products (e.g. gasoline, lubricating oil, kerosene, etc); (ii) predict when such products will arrive in strategic points (e.g. terminals, refineries, etc) located in different places; (iii) plan the best routes to transport a particular product; (iv) register the real data that inform when and which products arrive in some point; (v) compare real data with the predicted data; (vi) provide different types of reports and graphics to help on the analysis of different activities; and (vii) control when and which products are imported from or exported to other countries.

Aiming to identify a set of routes that gets to attend requests of customers, one of the modules of the application uses the paradigm of self-adaptive and self-test multi-agent system. This module looks for routes based on data provided by customer, and applies self-adaptation to meet the best route options.

4.1 Main Idea

The user of the multi-agent system provides a set of data to be used by the system to come up with the best routes following such requirements: (i) the *strategic point* (e.g. refinery) that must be attended, (ii) the product and amount desired, (iii) the type of the strategic point that should be the supplier of the product, and (iv) the maximum price that the customer is able to pay. The Manager agent receives such information and is responsible for checking the Route agents that can provide routes that have as supplier the same type of the strategic point informed by the customer. There are different types of Route agents: sea terminal, land terminal and refinery agent. Each type knows all the routes that have a specific strategic point as supplier of products.

The Manager agent contacts only the Route agent that knows the type of strategic point that can work as supplier of products as informed by the user. The Route agent

looks for routes that attend the data provided by the customer and send such information to the Manager agent. On the one hand, if a route is met, the data received by the Manager agent are forward to the customer. On the other hand, the Manager performs a self-adaptation looking for other routes that partially attend the request performed by the user. Figure 7 illustrates the idea of the system. Note that each Route agent reads a part of a corporative database in order to recover routes related to different types of strategic points.

4.2 Agents that Perform Self-Test

As stated before, the Manager agent is the responsible for performing a self-adaptation when the first Route agent does not provide any route that attends the request performed by the customer. With the aim to adapt its behavior, the Manager agent uses the control loop composed of the six activities described in Section 3.1.

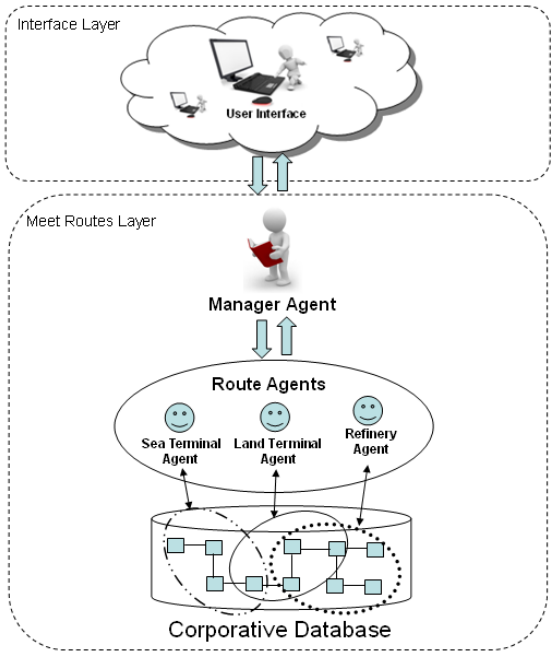

Figure 7. Petroleum control system

In the *collect* activity the agent collects data provided by the user and the last routes met by the Route agent.I It is possible to meet routes that can attend partially the request performed, in special the desired amount of a specific product and the cost that the customer is available to pay. Next, in the *analyze* activity the agent uses a forward chaining algorithm (Bigus et al., 2002) to meet the reason of the fail. Three main reasons were defined: (i) communication error between the agents of the system, (ii) one of the user requirements could not be attended (such as type of supplier, product, amount or cost), and (iii) any other execution error. The met diagnosis is then provided to the *decision* activity.

The *decision* activity receives such diagnosis and uses it to choose another behavior that can solve the problem. In order to meet such solution, a case-based reasoning algorithm (Amodt and Plaza, 1994) is used. Such algorithm finds the solution to a problem based on similar cases that relate problems and solutions. For example, in case no route is found because there is not any strategic point of a given type that can be the supplier

of the amount of product desired, the agent self-adapt its behavior to look for routes that include other types of strategic points to supply the product.

The *test* activity is initiated in the following. This activity executes a set of test cases respecting the control-flow, tests and data, defined in the CFL, TL and DL files, respectively. Different test cases are executed, such as:

1) One of the most simple test cases only verifies if the communication between the agents is possible;
2) Another test case verifies if connections with the database used by the system are being performed successfully;
3) The most interesting test verifies if a Route agent is recovering the correct set of routes from a database considering the strategy represented in the chosen behavior by the *decision* activity. The input data and the output assertions are defined in the DL file. Besides, the data defined in the DL file is not committed in such database in order to avoid the loss of stored data.

After executing the tests, the *validate* activity analyzes the log file that describes the result of each test case executed. Based on such information, the activity decides if the chosen behavior is adequate or not. If not, another behavior must be chosen and the *decision* activity is re-executed. However, when the tested behavior is validated, the *effect* activity is activated.

Figure 8. Example of map generated from the system.

When the *effect* activity is executed, the agent is reconfigured in order to use the chosen behavior. Figure 8 illustrates an example of a map generated with routes that attend a terminal in the state called Pará (PA) in Brazil.

5 Conclusion and Future Work

This paper presents the Self-Test Framework that extends JADE, a FIPA compliant framework developed in Java to implement multi-agent systems, in order to apply the

self-test concept. The main entities responsible for representing such concept are the *test* and *validate* activities, which can be reused in different self-adaptation processes. Thus, the main idea of this proposal is to evaluate, before adapting the agent behavior, if such behavior contemplates the environment/system requirements.

Aiming to demonstrate the use of our approach, we have used it in an industrial system, developed to a petroleum company, responsible for identifying routes that attend products derived from petroleum (e.g. kerosene, gasoline, etc). If the system is not able to find a route that follows the exactly request made by a customer, the system self-adapts to meet a route whose characteristics are very similar to the ones requested by the customer.

Nowadays, we are in the process of developing a tool able to automatically generate the content of the TL, DL and CFL files based on test models created by using a UML Testing Profile (OMG, 2010). The test cases source code will also be generated, such as, the unit tests that use JUnit and DBUnit API. Beyond this, we are evaluating how much time developer spends on learning to use the ST framework and the impact that the self-test concept has in the performance of self-adaptive systems.

References

Amodt, A. and Plaza, E., Case-based reasoning: Foundational issues, methodological variations, and system approaches. In AI Communications, volume 7:1, pages 39–59. IOS Press, March 1994.

Bellifemine, F., Caire, G., Trucco, T., Rimassa, G., Jade Programmer's Guide, 2007.

Bigus, J. P.; Schlosnagle, D. A., Pilgrim, J. R.; et. al..ABLE: A toolkit for building multiagent autonomic systems. IBM Syst. J. 41, 3, 350–371, 2002.

Costa, A. D., Silvia, V., Lucena, C. J. P., JAAF+T: A Framework to Implement Self-Adaptive Agents that Apply Self-Test. In Proceeding of the 25th Symposium on Applied Computing (SAC 2010), Sierre, Switzerland, pp. 928-935, March 2010.

DBUnit Web Site, http://www.dbunit.org/, Last access at August 2010.

Denaro, G., Pezze, M., and Tosi, D., Designing Self-Adaptive Service-Oriented Applications. In Proceedings of the Fourth International Conference on Autonomic Computing. IEEE Computer Society, Washington, DC, 16, 2007.

Dobson, S., Denazis, S., Fernández, A., Gaiti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F., A survey of autonomic communications, *ACM Transactions Autonomous Adaptive Systems (TAAS)*, 223-259, December 2006.

Fayad, M., Johnson, R., Building Application Frameworks: Object-Oriented Foundations of Framework Design (Hardcover), Wiley publisher, first edition, ISBN-10: 0471248754, 1999.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, 1st.ed. USA: Addison-Wesley, 1995.

IBM, An architectural blueprint for autonomic computing. Technical Report., IBM, 2003.

JUnit Web Site, http://www.junit.org/, Last access at August 2010.

Karam, K. S., Landslide Hazards Assessment and Uncertainties, Thesis: Massachusetts Institute of Technology, 2005.

Kephart, J. O. and Chess, D. M., The Vision of Autonomic Computing. *Computer* 36, 41-50, January 2003.

King, T. M., Ramirez, A. E., Cruz, R., Clarke, P. J., An integrated self-testing framework for autonomic computing systems, Journal of Computers, Vol. 2, No. 9, November 2007.

OMG - Object Management Group, UML Testing Profile, version 1, http://www.omg.org/cgi-bin/doc?formal/05-07-07, Last access at August 2010.

Stevens, R., Parsons, B., and King, T. M., A self-testing autonomic container. In Proceedings of the 45th Annual Southeast Regional Conference (Winston-Salem, North Carolina). ACM-SE 45. ACM, New York, NY, 1-6, 2007.

Wen, C., Wang, L.-C, Cheng, K.-T, Yang, K., Liu, W.-T.., "On a Software-Based Self-Test Methodology and Its Application". IEEE VLSI Test Symposium, May 2005.

Wooldridge, M. and Jennings, "N. R. Pitfalls of agent-oriented development," Proceedings of the Second International Conference on Autonomous Agents (Agents'98), ACM Press, pp. 385-391, 1998.