PLeTs

A Product Line of Testing Tools

Summary

Acronym List

EDS - **E**xchangeable **D**ata **S**tructures
HSI - **H**armonized **S**tate **I**dentifier
MBT - **M**odel **B**ased **T**esting
PIS - **P**erformance **I**ntermediate **S**tructure
PLeTs - **P**roduct **L**in**e** of **T**esting Tool**s**
SMarty - **S**tereotype-based **M**anagement of V**ar**iabili**ty**
SNMP - **S**imple **N**etwork **M**anagement **P**rotocol
SPL - **S**oftware **P**roduct **L**ines
SSH - **S**ecure **Sh**ell
STD - **S**tructural **T**esting **D**ata
UML - **U**nified **M**odeling **L**anguage
XMI - **X**ML **M**etadata **I**nterchange
XML - E**x**tensible **M**arkup **L**anguage

1 Motivation

The cost of software testing is related to the number of interactions and test cases that are executed during the testing process. As it is one of the most time consuming and expensive phases of software development, MBT is a well-known approach to mitigate this problem by automating the process of generating test cases or scripts. However, even though several works on MBT were produced in the past years, the effort to design and develop MBT tools are made from scratch.

Based on that, it would be useful to define a single architecture that could be used to generate MBT tools to cover all the phases of the MBT process, *i.e.* a tool in which it would be possible to describe the system model, that would generate test cases/scripts, that would execute the test scripts and also compare the results. It would be even better if the test team could generate a testing tool for each different application or different testing techniques and then execute it over the same application. Furthermore, it is desirable that the test team reuse implemented artifacts (*e.g:* models, software components, scripts).

Other issue that the test team has to face is that in some situations it could be necessary to adopt another testing technology and consequently use a different testing tool to test an application. Moreover, the necessity to adopt a different testing tool could also be motivated by non-technical factors, such as cost, availability of a tool with more features, and market decisions. Furthermore, in these situations, it is usual that a large investment is spent in software licenses and to train the test team. Besides that, a lot of pressure is put over the test team to quickly learn the new technology and then explore all the tool features. Another issue is that, in some situations the testing team is already motivated to move from some testing approach to a MBT approach, but when they realize that the already purchased tools are useless to apply MBT, they give up.

To avoid these issues, during the design and development of a MBT tool, the *Build Model*, *Generate Expected Inputs*, *Generate Expected Outputs* steps should be designed and developed bearing in mind that an external tool (*e.g*, LoadRunner and Visual Studio) could be used to execute the *Run Test* step [ELFAR:01]. Thus, the test team could reuse the testing tools to apply MBT, reducing time and cost in the development of another MBT tool. Although the possibility of use an external tool to execute MBT is useful to testing community, this approach does systematize the reuse of software components already developed to implement each MBT step. Another point is that it is necessary to control and plan the variability between the components. Thus, in this context, it becomes valuable to design a set of MBT tools based on a Software Product Line (SPL). A SPL ensures the

variability and reusability of testing tools artifacts, thus decreasing costs and time to market. In the last years, there are some works that discuss software testing and SPL, but all of them focus on the testing of software product lines and do not investigate how the SPL concepts can be applied to support the development of MBT testing tools [OLIMPIEW:2005] [ENGSTROM:2011].

2 PLeTs

PLeTs is an SPL to automate the generation of MBT tools. These tools aim to automate the test activities of MBT processes [ELDER:2010], [SILVEIRA:2011], [COSTA:2012]. The MBT process automates the generation of test cases and/or test scripts based on the system model. As presented in [ELFAR:01] the MBT main activities are: *Build Model*, *Generate Expected Inputs*, *Generate Expected Outputs*, *Run Tests*, *Compare Results*, *Decide Further Actions* and *Stop Testing*. Therefore, the MBT tools derived from PLeTs support some or all of the MBT activities. They accept a system model as an input, generate the test cases/scripts (*Expected Inputs/Outputs*), execute the test scripts and then compare the results. It is important to note that PLeTs was designed to generate scripts based on a template for a specific testing tool. As a consequence, a testing team could use its legacy testing tool to apply MBT, reducing effort and investment. After that, the PLeTs product runs the testing tool, loads the generated scripts, starts the test (*Run Tests*) and compares the results (*Compare Results*). Thus, the main PLeTs goal is the reuse of SPL artifacts (*e.g.* models and software components) and use testing tools to make it easier and faster to develop a new MBT tool. Figure 1 presents the current PLeTs feature model, which is composed of five main features:

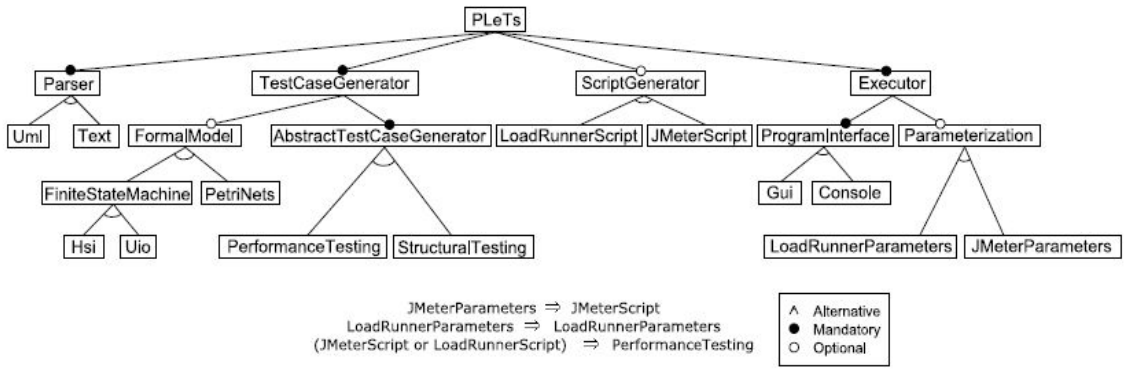

Figure 1. PLeTS feature model

- *Parser*: automates the *Build Model* step in the MBT main activities. It is a mandatory feature with two child features, *UML* and *Text*. The former extracts information from UML models and the latter extracts information from a textual file;
- *Test Case Generator*: represents the *Generated Expected Inputs* step in the MBT main activities. It is a mandatory feature with two child features: *Formal Model* and *Abstract Test Case Generator*. The former has two features: *Finite State Machine*, which has the child features *HSI* and *UIO*, and *Petri Nets*. The latter has two child features *Performance Testing* and *Structural Testing*. These two features receive the generated test sequence and create the abstract testing sequence to each testing level.
- *Script Generator:* is an optional feature that is used to convert abstract test cases into scripts for a testing tool responsible to perform the actual execution of the System Under Test (SUT). *Script Generator* has five child features*: LoadRunner Script, Visual Studio Script, Emma Script, JaBUTi Script* and *JMeter Script* that are used to consolidate the abstract test case in scripts to a specific testing tool;
- *Executor*: represents the product interface and the test execution. This feature also has two child features: *Product Interface* and *Parameterization*. The former defines the product interface and is composed of two child features: *GUI* to graphical interfaces and *Console* to command line execution. The latter has five child features: *LoadRunner Parameters, Visual Studio Parameters, Emma Parameters, JaBUTi Parameters* and *JMeter Parameters*. These features are implemented to start the testing tool and to run the test scripts. After that, the testing tool (*e.g*, LoadRunner) collects the test result, shows the results and compares them with test oracles.
- *Monitoring:* this feature has two child features: *SSH* and *SNMP.* The former feature define that the SUT monitoring is done using the SNMP protocol. The *SSH* feature define that the SUT monitoring is done using the SSH protocol. Both of the features are used to monitoring the SUT and then show the results to the testing engineer. It is important to highlight that this features should be used when some protocol, operation system or technology is not supported by the load generator used by the MBT tool generated from PLeTs.

It is important to mention that, even though our current feature model has a well-defined number of features, this model can, and will be expanded to include new features. With regard to Figure 1, there are several dependencies that are denoted by propositional logic between features. For instance, if feature *Executor* and its child feature *LoadRunner Parameters* are selected, then the feature *Script Generator* and its child feature *LoadRunner Script* must be selected as the generated tool is not able to execute tests with no test script. Furthermore, it is important to note that the PLeTs feature model can be

extended to support new testing techniques or tools by adding new child features to its main features. For instance, if one adds new features for the SilkPerformer testing tool, new child features for the *Script Generator*, *Product Interface* and *Parameterization* features must be included.

3 PLeTs Architecture and Implementation

In order to implement PLeTs, we have used a component-based mechanism to develop each feature of our feature model [CERVANTES:2006]. The main idea of this strategy is to plug a component without requiring special explicit intertwined configuration of a specialized software engineer. The combination of components, one for each desirable feature, generates a product. Thus, a MBT tool derived from PLeTs is assembled by installing a set of selected components on a common software base. We chose this approach to generate the PLeTs products because it presents several advantages, as the high-level of modularity and decoupling between the base application and components. Furthermore, a component-based SPL presents other benefits as, for instance, the components can be developed independently and geographically distributed, reducing time to market and costs (Cite).

To manage the dependencies among components and represent the variability in PLeTs, we apply the SMarty (**S**tereotype-based **M**anagement of V**ar**iabili**ty**) approach [OLIVEIRA:20102]. SMarty is composed by a UML profile and a process for managing variabilities in a PL. The SMarty profile contains a set of stereotypes and tagged values to denote the PL variability. The SMarty process consists of a set of activities that guide the user to trace, identify, and control variabilities in a PL. Figure 2 shows the PLeTs component diagram in accordance to SMarty that reflect our feature model shown in Figure 1.. The main PLeTs components are: *Parser*, *Test Case Generator*, *Script Generation, Executor* and *Monitoring*.

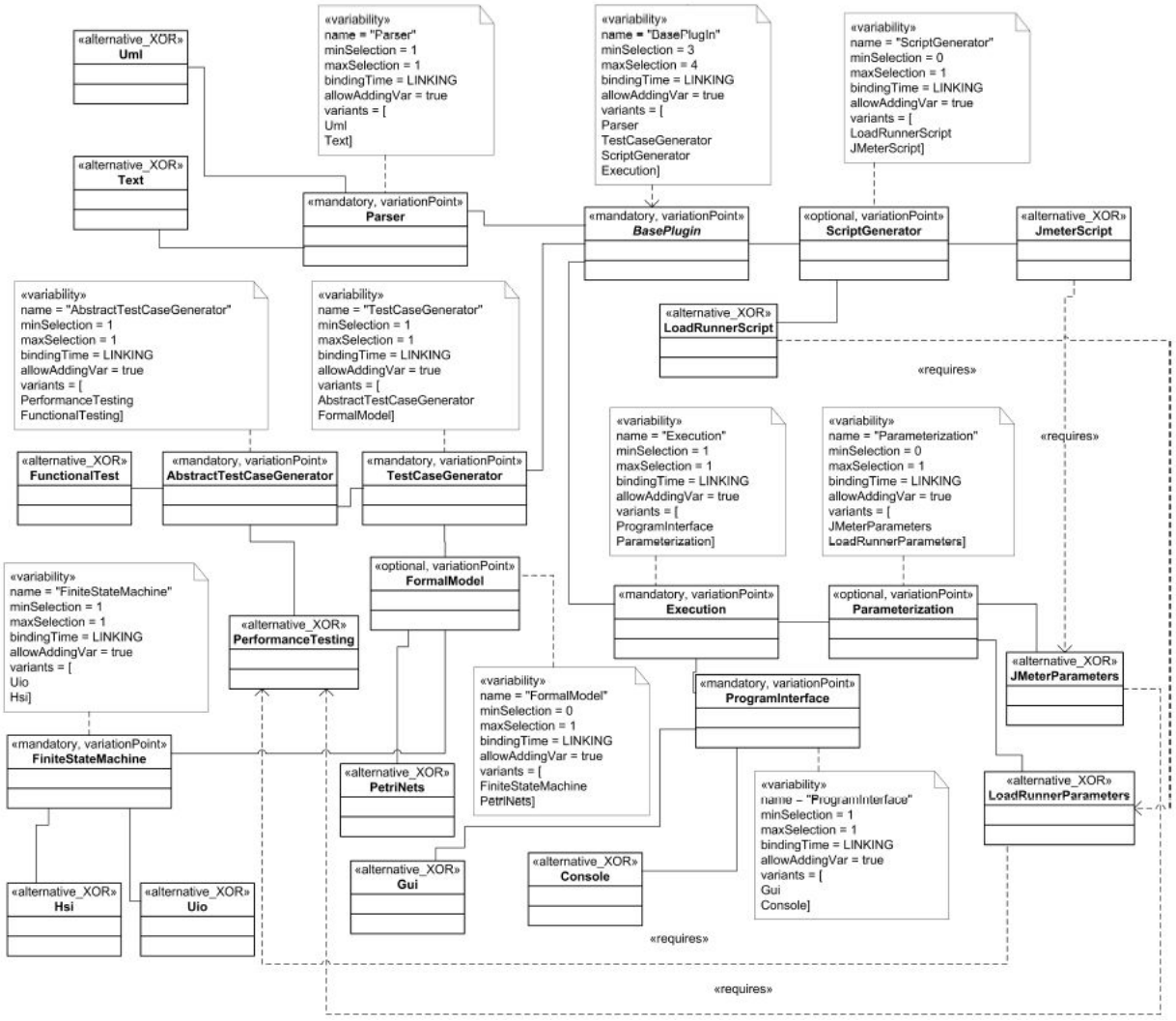

Figure 2: PLeTs UML component diagram with SMarty notation.

The following sections demonstrates the main PLeTs components, implementation details and specific component information. This information will be presented in the following format: (i) component's behaviour and description; (ii) an activity diagram exemplifying the behaviour; (iii) a snippet of code that implements basic component functionalities and (iv) a detailed description of the code, with implementation details and specific code notes.

It is important to note that some rules are implemented by all components presented in PLeTs, but others are specific for some component. For means of organization, code readability, platform specific dependencies and PlugSPL dependencies, the following rules must be respected:

- Every component is implemented in separate software artifacts, that is, each component is an independent software, which should work inside the PLeTs environment as a module;

- Since the PLeTs platform code is implemented over .net framework, each component must be compiled as a separate project;

- Component projects must contain a valid class name. That name must be the base namespace inside the assembly and a class with the same name must be defined inside that project. That class will be used by PLeTs to initialize the component;

- Every PLeTs component must implement a *Run* method, which receives the *map* object as parameter (the object *map* - see Figure XX). That map parameter is a Hashmap containing the product configuration (the set of components), generated by the PlugSPL /cite(). This map is used by PlugSPL to link components at runtime and should not be manipulated by components directly.

All these rules will be explained with more details in *General Component Development* section (Section X).

3.1 Parser

Parsers are components used by PLeTs' products to extract data from the system models. These data are used to build an specific data structure which is used to the others components of the PLeTs products to generate and execute the test scripts and scenarios. As represented in the Figure 1, each MBT product derived from PLeTs must have a Parser component attached to it. It allows the PLeTs products extract information from given models and build the *Exchangeable Data Structures* (EDS) at run time. That is, a Parser receives an specific model as input and generate the EDS as an output. It is important to note that the input model could be a text file, a XMI file, or any format of file - since it represents a system model. The result of the parsing process is an EDS filled with data extracted from the system model.

3.1.1 Parser Implementation

Since the Parser component expects a system model as input and produces an EDS as output. The produced EDS is sent to the *Test Case Generator* component.

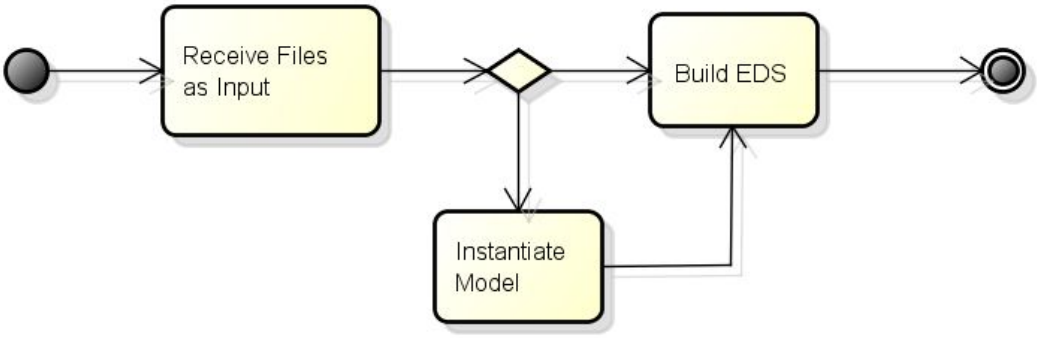

Figure 3: Parser activities

The Figure 3 present the basic activities that a PLeTs's Parser must implement. Thus, every parser component must to accept a system model as an input (activity *Receive Files as Input*), instantiate the model structure (activity *Instantiate Model*) and then build the EDS (activity *Build EDS*)

Furthermore, to develop a valid PLeTs's parser component is mandatory to implement an interface called *IParser*, that is defined in the *PLeTs.PLeTs* namespace, which is inside the *PLeTs.dll* - the platform's core component. It is also mandatory to implement the *Run* method, since its method is used by the platform to start the component. As presented in the Figure 3, the method must receives the component map as parameter (for more details about the *map* object see Section XX) and must support the following activities: i) *Receive Files as Input*, where the data is extracted from the given files and validated; ii) *Instantiate Model*, where the system model is instantiated to simplify the creation of *EDS*; iii) *Build EDS,* the extracted data from the *Instantiated Model* and convert it in an EDS (PIS or STD) and serialize it using the *XmlSerializer* class (.NET framework). The code snippet below show an example of how a parser is structured.

```
using PLeTs.PLeTs;
public MyStdParser: IParser{
      public Object Run(Hashtable map){
            Object info = ParseInfoFromModel();
            Object std = GenerateStd(info);
            Object serialStd = SerializeStd(std);
            return serialStd;
      }
      internal Object ParseInfoFromModel(){ ... }
      internal Object GenerateStd(){ ... }
      internal Object SerializeStd(){ ... }
}
```

Simple Parser implementation

After the method execution (see Figure XX), the EDS is ready to be sent to the *Test Case Generator* component. The code snippet below illustrate the interaction between a *Parser* and a *Test Case Generator* component from the *PLeTs.dll* component view.

```
public Program{
	public static void Main(){
		//...
		IParser p = Activator.CreateInstance("MyStdParser");
		Object o = p.Run(map);
		TestCaseGenerator g = new TestCaseGenerator();
		g.Run(map, o);
		//...
	}
}
```

Parser and Test Case Generator interaction

It is important to highlight that regardless of what is the input format accept by the parser, a method *Run* must be implemented and its output must to be a *EDS*. However, the source code inside these methods has no conventions or rules - the informations presented in this section are just a guideline and may not be followed if not suitable. Another point is that the parser component is developed based on our expertise, so the current *EDS* can not store all information needed to produce MBT testing tools for a specific domain or technology.

In the next sections we will present two different types of *EDS*, the *Performance Intermediate Structure* and the *Structural Test Data.*

Bear in mind that these structures support in a limited way, only performance and structural testing. Thus, if someone else want to develop a tool to apply a different testing technique (such as functional testing), it is necessary to develop its own structure.

3.1.2 Performance Intermediate Structure - PIS

The *Performance Intermediate Structure* is a structure designed to store the informations extracted from the system models and that is needed to generate performance test scenarios and scripts. In the Figure 3 is depicted the PIS class diagram,

where the main class is the class "TestSuit", which is associated with a collection of objects of the class "Scenario." The performance test scenario is the artifact that sets a number of test parameters, e.g. number of virtual users represented by the "Population" property. Moreover, the class "Scenario" has a collection object of "TestCase" classes. This class, in turn, has a list of requests (class "Request") and transactions (class "Transaction") that are executed by the test case. Each "Request" can be associated with one or more parameters (class "Parameter"), and may have one or more transactions (class "Transaction"). The performance testing transactions are used to isolate the processing of a set of requests. Furthermore, the transactions definition aims to facilitate the identification of bottlenecks on the system. Moreover, transactions are also useful for TPS metric, which can be used to evaluate the system acceptance criteria (e.g., a Service Level Agreement). There are other classes: "Host" which is associated with the test scenario, determining the server ("Host") in which is SUT hosted, moreover to other servers that may be associated with SUT (e.g., database); finally, the classes "Counter" and "Metric" which determines the performance counters and the associated metrics to each counter that will be related to objects of class "Host".

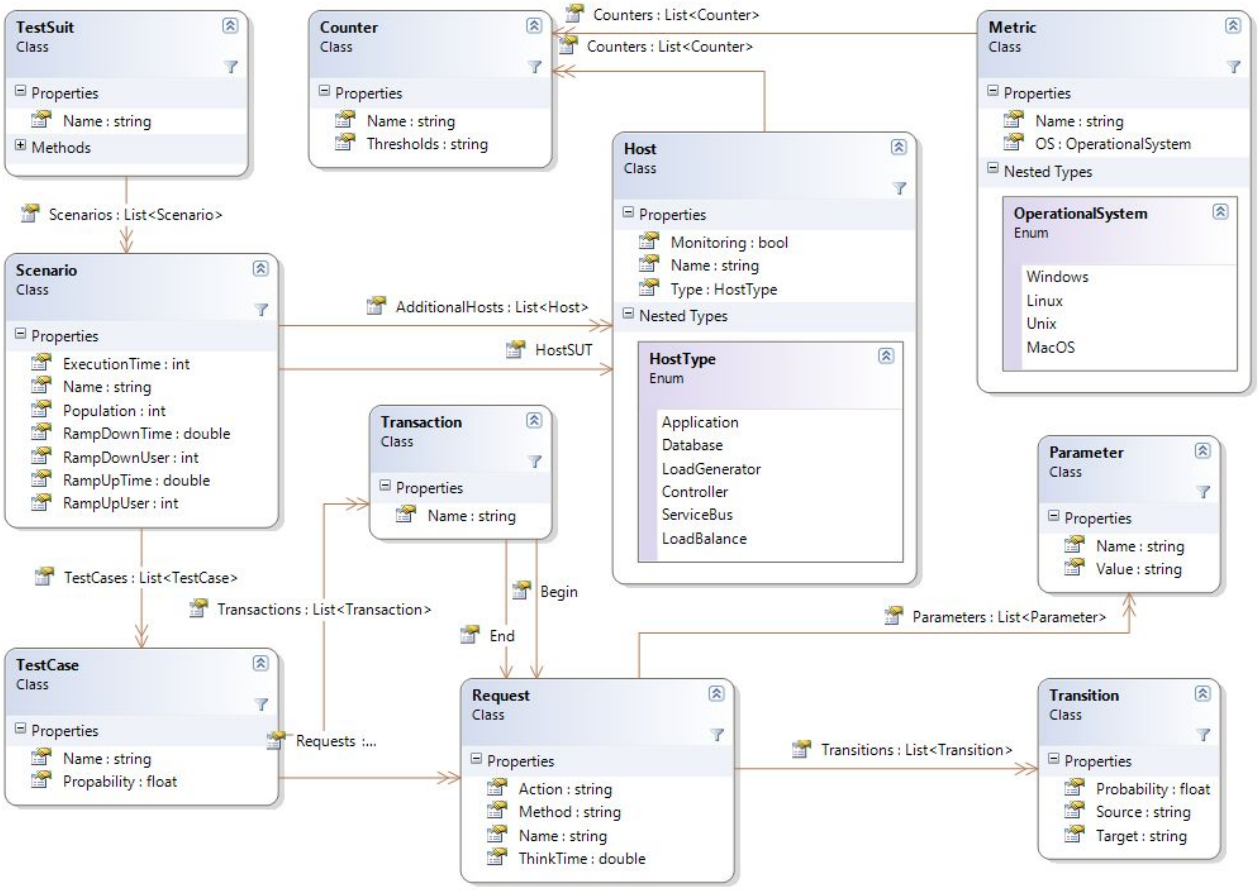

Figure 3: Performance Intermediate Structure UML class diagram.

3.1.3 Structural Test Data

Structural test data (STD) is similar to the performance testing intermediate structure. However, STD is a structure in memory that contains information needed to automate structural test case generation. The information stored on STD is extracted through parsing data from a XMI file, which is generated from UML sequence diagrams. We choose UML sequence diagrams because they can be used to represent low-level information and the order that methods of an application are executed. Therefore, these diagrams contain information about the components that will be tested, *e.g.* parameters (name, type) of classes and methods, as well as, each method return type.

The STD class diagram (see Figure 4) consists of five classes. At the highest level of STD hierarchy is the *Structural Test Data* class. This class is associated to a collection of objects of "UmlClass" and "UmlTag" classes. "UmlClass" has information about the classes that will be tested and it consists of a name and a set of methods ("UmlMethod"). "UmlMethod" has information about a specific method to be tested. This class consists of a set of attributes, *e.g.* identifier, name, return type and a collection of parameters ("UmlMethodParam"). "UmlMethodParam" has information related to parameter values of a method and the type of a specific parameter. To finish the STD class diagram description, the class "Tags" is used to store information about the classes to be tested, their classpath and the technology (structural testing tool) information, which is used to generate and execute test scripts.

The STD is the input of PLeTs "Test Case Generator, which applies a random test data generation technique [REF] in order to select a set of specific parameter values for the methods to be tested. This information is used to, automatically, generate structural test scripts (*Test Driver* files) that can be executed by different structural testing tools.

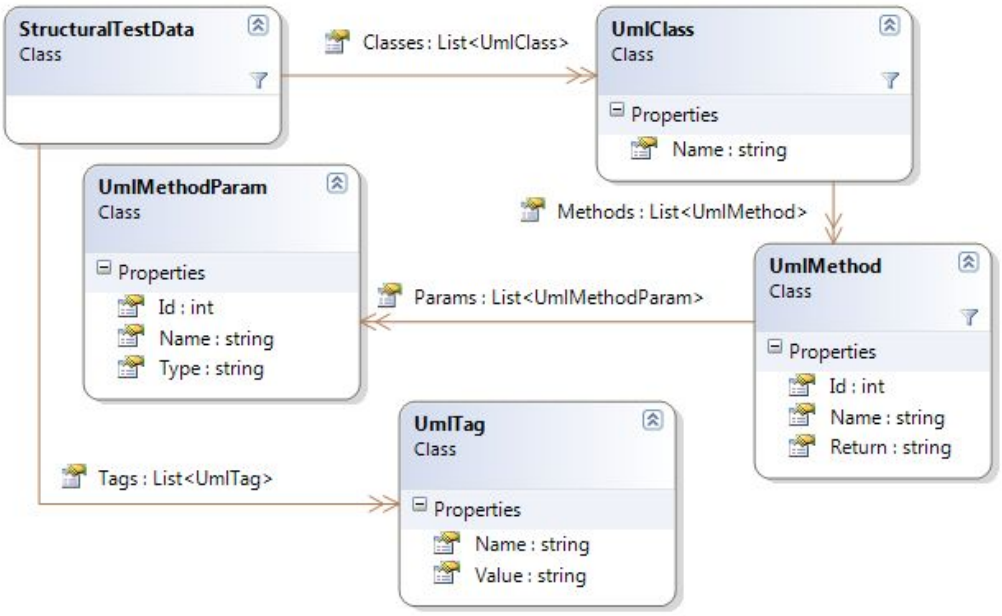

Figure 4: Structural Test Data UML class diagram.

3.2 Test Case Generator

The Test Case Generator component is design to generate the test sequences and the abstract test cases. This component uses the EDS as an input and then based on that generate the abstract test case as an output. Thus, this component orchestrate the generation of abstract test case that is used to generate the technology-specific test scripts (e.g., LoadRunner and Jmeter). It is important to note that the abstract test case has a different structure and informations for each testing technique. Thus, to be used for the PLeTs structural testing tools we design a structure called *abstract structure*. Likewise, for the performance testing tools we design a structure named *Abstract Test Case*.

To generate the *Test Drive* or the *Abstract Test Case,* the *Test Case Generator* component needs a *Sequence Generator* component attached to it in order to generate random (structural testing) or coverage data sequences (performance testing). For this purpose, structural testing uses a *Random Test Data Generator* component, while the performance testing uses a specific method in combination with a formal model.

Figure 5 illustrates the internal data flow of *Test Case Generator* component. The default PLeTs *Test Case Generator* receives the outputs from the *Parser* component, sends it

to a *Sequence Generator*, gives the *Sequence Generator* output to the *Abstract Test Case Generator*, attaches the *Abstract Test Case Generator* output to the PIS or STD and sends it to the *Script Generator* component.

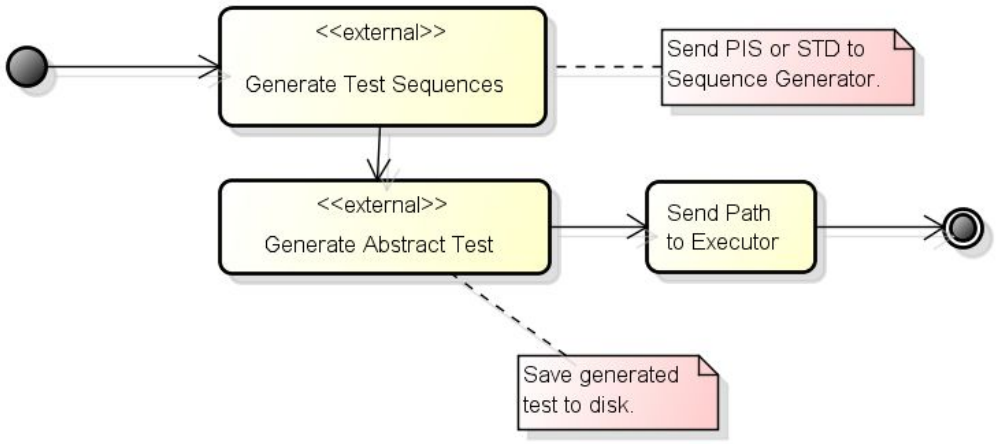

Figure 5: Test Case Generator data flow.

3.2.1 Test Case Generator Implementation

The *Test Case Generator* component does not implement any data manipulation over the STD or PIS. Instead, it controls the flow of information through the four main components attached (*Parser, Sequence Generator, Abstract Test Case Generator and Script Generator*).

The *Test Case Generator* component receives as input (parameter) an Object "o" which is sent to an instance of *ISeqGen* (the *Sequence Generator* component) named "sg". The results of the sequence generation is returned to the *Test Case Generator*, and then sent to the *Abstract Test Case Generator*, which must generate a *Test Drive* or a *Abstract Test Case,* save it to disk and send its path to the *Test Case Generator* component. Thus, the *Test Case Generator* component outputs the path of generated files to the *Script Generator* component. This behaviour is illustrated in the following code snippet.

```
public TestCaseGenerator{
      public Object Run(Hashtable map, Object o){

            ISeqGen sg = Activator.CreateInstance(sequenceGeneratorTypeName);
            Object sgResults = sg.Run(o);
```

```
            IAbstrTestCaseGen atcg = Activator.CreateInstance(atcgTypeName);
            Object actgResults = actg.Run(sgResults, o);

            return actgResults;
      }
}
```

An example of a Test Case Generator implementation

The structures which flow through the *Test Case Generator* do not need to be unserialized, since the *Test Case Generator* does not manipulate that data directly - but attached components must do unserialization internally.

Since sequence generation is a way to simulate users behaviour and system calls into the test project, in order generate useful sequences, some sequence generation technique must be implemented to obtain that sequences (ref 2 or 3 technique of generate sequence - random, methods. etc). The generation of sequences for performance testing could uses formal models, such as *Finite State Machines* (FSMs)(ref) and *Petri Nets* (ref). These formal models could be used in combination with a *Sequence Generation Method,* such as *HSI* (ref) or *UIO* (ref) (both for *Finite State Machines*). On the other hand, the structural testing uses a *Random Test Data Generation Technique* (ref). Although there are different techniques to generate sequences, any component that generate testing sequence can be attached to PLeTs, since it receive as an input the STD or PIS structures and generate the *Abstract Test Case Generator* as an output . The Figure XX present a code snippet that illustrate an example of a sequence generator component. It is important to mention that any implementation of a *Sequence Generator* must generate an output containing the generated sequences, no matter which technique is used to generated it.

```
public MySeqGenerator : ISeqGen{
      public Object Run(Hashtable map, Object o){
            Object x = GenerateSequence();
            return x;
      }
}
```

An example of a Sequence Generator implementation

3.2.2 Formal Models and Sequence Generation Methods

To simulate user behavior over a system, the *Test Case Generator* component uses a formal model in combination with a specific sequence generator method. Both, *Formal Models* and *Methods* are implemented in the same component (one component per combination). An example of combination is the *Fsm-Hsi* component, that implements the *HSI* method, that receive a *Finite State Machine* as an input. To obtain the sequences that simulate the user behavior inside a web system, the component generates a sequence of *requests objects*, which represents the browser requests to the server (emulating the user interaction with a web page).

In order to generate the test sequences, the PIS is sent as reference from the *Test Case Generator* component to the *Sequence Generator* component. The *Sequence Generator* component is invoked by the *Test Case Generator* component one time for each test case that is in the PIS. So, each *Test Case* object of the PIS is converted into a *Finite State Machine*, where the machine states are the representations of application pages and machine transitions are representations of the requests to the SUT. Thus, the number of finite state machines generated by the *Sequence Generator* component is equal to the number of test cases in the PIS. The requested URLs (user actions) are used as *inputs*, working as *input alphabet* in the FSM. After the generation of the FSM, it is used as an input to the sequence generator method in order to generate the test sequences. Thus, the generated sequences are attached to PIS, and then sent back to the *Test Case Generator* component, which is responsible for send it to the *Script Generator* component.

3.2.3 Random Test Data Generator

In order to select the test cases for generating test scripts, the PLeTs *Test Case Generator* receives as input the STD and a XML file (*Test Data*) with different parameter values for all classes and methods of the SUT. Based on that, the PLeTs *Test Case Generator* applies a random test data generation technique [REF] under the parameter values presented on *Test Data* and only for the classes and methods described on STD. This random technique is applied in two different ways: one for selecting the amount of test cases for each method to be tested and another for selecting which test cases will be chosen.

After applying that technique, the PLeTs *Test Case Generator* also produces the abstract structure. The abstract structure consists of a text file that describes, in a sequential and non technology-dependent format, the entire data flow of the classes and methods selected after applying the random technique. The abstract structure is divided into three groups:

- Technology Configuration: specifies the information about the technology that will

be used for the test execution;

- Test Configuration: specifies the path of the classes that will be tested, a list of imported classes (import, package) and the path of SUT's library;
- Sequential Flow Configuration: defines the sequential flow of the methods and classes that will be tested.

The abstract structure is the input for the PLeTs *Script Generator* component. This component generates the *Test Driver*, which is used for executing structural testing by different testing tools.

3.3 Script Generator

The *Script Generator* component is designed and developed to consolidate the abstract test cases. In case of performance components, the *abstract test cases* are consolidated in *scripts* and *scenarios* for some performance testing tool, e.g., LoadRunner. For structural components, the *abstract test cases* are consolidated in *Test Driver* files/scripts for some structural testing tools, *e.g.* JaBUTi.

To consolidate the *abstract test cases* to performance scripts and scenarios for some performance testing tools, the *Script Generator* component concretize it in executable scripts for a performance testing tool. As the *abstract test cases* is a sequence of user activities, the component use each sequence to generate one script for an specific performance tool (technology-specific file formats). The current version of the *Script Generator* component generate scripts and scenarios using as base a scripts template. To perform the conversion from the generated abstract test cases to the tool scripts and scenarios format, the Script Generator uses a set of file templates, that are copies of scripts, scenarios and others files generated from the tools containing placeholders. That placeholders are filled with the data generated by PLeTs components. Thus, the component and consequently the PLeTs products are able only to use performance tools that has support to import some script file/template.

On the other hand, to consolidate *abstract test cases* to structural test for some structural testing tools, the *Script Generators* receives the *abstract structure* as input. Based on this structure, the *Script Generators* builds the *Test Driver* file. After that, the *Test Driver* needs to be compiled in order to generate a class file. This is due some structural testing tools perform structural analysis only from the bytecode of the classes that will be tested. After compiling the *Test Driver*, the generated class is used for executing structural tests by different testing tools. The *Script Generator* behaviour is illustrated in Figure 6.

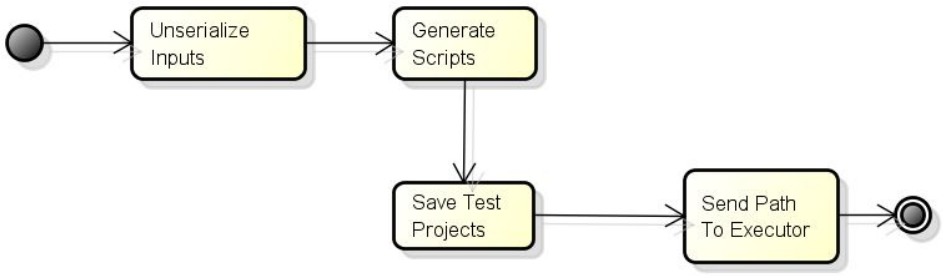

Figure 06: Script Generator behaviour.

It is important to mention that the current script generators components export performance test projects to LoadRunner, VisualStudio and JMeter tools, and structural *test drivers* (scripts) for JaBUTi and EMMA tools.

3.3.1. Script Generator Implementation

To develop a validate *Script Generator* component, it must execute four main activities (see Figure XX?): (i) unserialize *Test Case Generator* output; (ii) generate test project; (iii) save test project and; (iv) send the directory path where the test project was saved. The component must also implement the `IScriptGen` interface, used for automation purposes.

```
public MyScriptGenerator : IScriptGen{
      public Object Run(Hashtable map, Object o){
            //input deserialization
            //generate test project
            //save test project to disk
            return pathToTestProject;
      }
}
```

An example of a Test Case Generator implementation

It is important to highlight that some test projects (such as *LoadRunner* projects) contain script paths and other path information inside generated files. That references may cause problems when renaming or moving the project through files systems.

3.4 Executor

In the *Executor* component is configured the launch and the parameterization of the external testing tools, e.g., LoadRunner and JaBUTi. This component stores command line options, file paths, directory entries and others informations used by the PLeTs product at runtime. Some information about where the testing tool is installed on the system (directory, executable name or command) and the generated scripts location (saved by the script generator component) could be requested during the execution of the PLeTs product. After acquiring the information, the external tool is ready to launch and execute the generated testing projects.

3.4.1 Executor Implementation

The first step to develop an Executor component is to know how to call the external testing tool that will be used to execute the test scripts. The next step is to implement the component. Thus, to implement the component is mandatory that it main-class implement the *IPLeTsExec* interface. This interface has no mandatory methods because it is used to help the product orchestration inside the PLeTs. The code in the Figure xx exemplify how the LoadRunner tool is called from the *LoadRunner Parameterization* component.

```
public LoadRunnerParameterization: IPLeTsExec{
      public Object Run(Hashtable map, Object o){
            String testsPath = (String)o;
            String tool = "C:\HP LoadRunner\wlrun.exe";
            ExecuteApplication(tool + " -a " + testsPath);
            return null;
      }
}
```

An example of an Executor component implementation

The method presented in the line xx receives the map as a parameters, used by the platform (for more details see the Section XX) and an object (named “o”), which contains the path where the test project was saved. The method `ExecuteApplication` might change between environments, and must be implemented through platform API.

3.5 Monitoring

The monitoring component is designed to monitoring the performance testing

results from the SUT. These results will be collected while the performance external tool (e.g., LoadRunner) is generating load over the SUT. Currently in the PLeTs, there are two performance monitoring components: SSH and SNMP. Both components monitoring the system in a similarly way: it open a SSH connection with the SUT in order to execute operating system commands or SNMP monitoring commands while the test is running. Thus, the SSH component execute Linux operating system commands, such as, *free*, *top* and *disk* and the SNMP component use commands provided by SNMP protocol \cite().

Since the monitoring components uses a SSH connection to access the SUT, thus is need an user account in the SUT. The account privileges shall be configured according to the commands that the monitoring component will execute. For instance, when the component is configured to monitor the *Memory Usage* metric, the command *free* is send through an SSH session. The user configured in the monitoring component must to have access (privileges) to execute that command. Moreover, the SSH server must be running in the SUT and must be configured to answer the monitoring component requests. At the client side, a SSH client must be installed in order to support the SSH component operations. The SSH client can be a library or an external tool. Currently, the SSH component uses the *Renci.SshNet.dll* which is necessary to establish SSH connections from Windows systems.

To the SNMP component is required a system user that must be able to execute SNMP commands and the SNMP service must be active into SUT. Moreover, the component requires the *net-SNMP* component, which is necessary to establish SNMP connection over Windows systems.

Finally, both component might works together the *Windows Perfmon*. The *Windows Perfmon* is an Windows native tools used to monitor the performance of a system. Thus, the components adds performance metrics to the *Perfmon* and then send a command over the SSH connection, collect the result data, parse it and updates the *Perfmon Monitor*. The Perfmon Monitor is also used by some commercial performance tools (such as LoadRunner, Visual Studio, etc) to retrieve data from the system.

3.5.1 Monitoring Implementation

Since the monitoring component no exchange data from/to other PLeTs components, it has no restrict implementation rules, so it must just follow the *general component development* guidance in order to work inside a PLeTs product.

Figure xx shows how a generic monitoring component must be implemented.

Basically, the `StartMonitoring` method must set the *Perfmon* metrics, configure the needed tools inside SUT machine and start the monitoring before the performance testing tool start its load generator.

```
public MyCustomMonitor: IMonito{
      public Object Run(Hashtable map, Object o){
            Hashtable info = (Hashtable)o; //SUT info
            StartMonitoring();
            return null;
      }
}
```

Monitoring component implementation

4 Product Generation Overview

In this section we will present a general guideline about PLeTs component development. As present above, a PLeTs product is a set of components which generates test projects for many testing tools, using the MBT approach. Each product is a different combination of components derived from the PLeTs. To guide the component selection, PLeTs is modelled using a UML component model, which is used to adds *exclude* and *requires* between components at configuration time. The selection of the components is made by the PlugSPL environment \cite{}. This environment support the generation of products from a component-based SPL. In other words, the PlugSPL environment is used to support the product generation from PLeTs. The environment is composed by three modules: *SPL Design*, *Product Configuration* and *Product Generation*.

The *SPL Design* module aims to design a FM by either creating it from scratch or importing a pre-existing FM from SPL tools. The *Product Configuration* module is responsible for checks the system consistency and generates the target system architecture. The *Product Generation* module takes the target system architecture as input, retrieves the component from the component repository, creates ``glue code'' for the components and packages it in an executable file.

4.1 General Component Development Guidelines

In this section we will described how the PLeTs components must be developed for work properly with PlugSPL.

Basically, the PLeTs components must be a .NET based-component and must has the same name as the component model in the PlugSPL. For instance, the *Uml* component should be implemented as a component named *Uml* (in the *Uml.dll* assembly). That component must to contain a namespace named *Uml* (same name as the feature), which in turn must to contain a class named *Uml.* Note that the *Uml* class must to contain a method called *Run*, which will initialize the component.

```
namespace Uml;
public Uml: IParser{
	public Object Run(Hashtable map, Object o){
		//code
	}
}
```

Example of UML component implementation

The Figure xx present the *Uml* parser implementation, where the namespace is called the same as the class. The class name is the same as the component name as well, and it must be followed by every component in the PLeTs. The *Run* method is the method which will trigger the component logic, and its called by the ancestor components (defined in the SPL component diagram). Additionally, the *Uml* class implements the *IParser* interface, setting the *Uml* component as an implementation of the parser feature, mapped in the platform.

To ensure that each component will work in accordance with its models, PlugSPL uses the interface definitions to allocate each component into the right place at execution time.

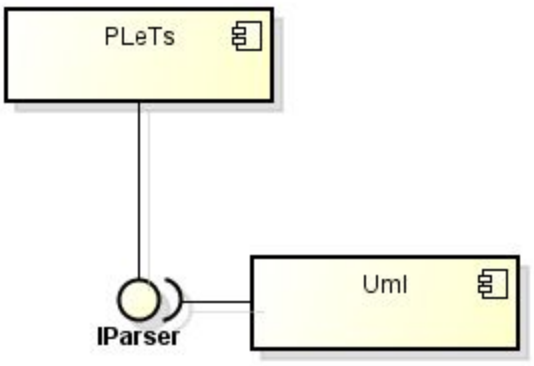

Figure 14. Uml implementing the IParser interface

Figure xx presente that the `IParser` interface is available to validate Parser implementations. Moreover, there are more four interfaces available to validate other components : `ITestCaseGen`, `IScriptGen`, `IPLeTsExec`, *IMonitor* (see Figure xx).

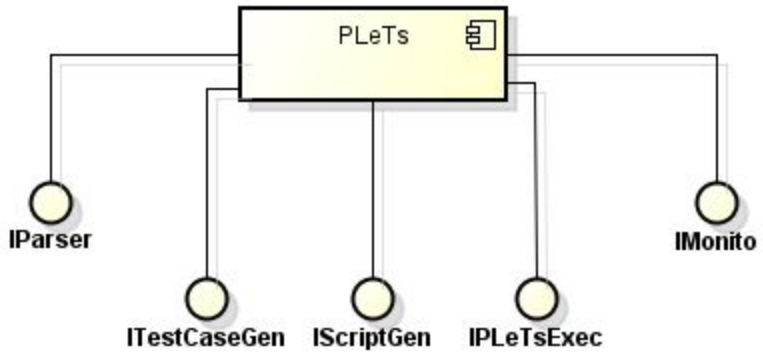

Figure 15. PLeTs platform interfaces

Like PLeTs, other components have interfaces to be implemented by other components - that is the way how PLeTs links the components. Note that leaf features has no interfaces to be implemented, because it represent no extension points in feature model. The only component which do not need to implement any interface is the PLeTs component (also called core component, or platform).