

Performance Analysis Module for UML Diagrams Guide

February 25, 2005

Contents

1	Introduction	3
2	Getting started	3
2.1	ArgoUML from CVS	3
2.1.1	Getting ArgoUML	3
2.1.2	Environment variables	4
2.1.3	Building ArgoUML	4
2.1.4	Building the Performance Analysis module	5
2.2	ArgoUML from binary distribution	6
2.2.1	Getting ArgoUML	6
2.2.2	Running the Performance Analysis module	6
3	Source code structure	6
3.1	Overview	6
3.2	Scalability	7
3.3	Directories structure	8
4	Components description	15
4.1	Model Editor	15
4.1.1	Plugging-in to ArgoUML	15
4.1.2	Generation of the XMI File	15
4.2	The Model Configurer	16
4.2.1	The Tag Value Language	18
4.2.2	Configuration and model annotation	22
4.2.3	The GRM Grammar	22
4.3	UML Validation	23
4.4	Model Processor	25
4.4.1	Queries the module can answer	25
4.4.2	Model Convertor	26
4.4.3	Model Analyzer	27
4.4.4	Results Convertor	27

5	Execution Flow	27
5.1	PAModuleAction	27
5.2	Step1: User selects query	28
5.3	Step2: Model Editor	29
5.4	Step3: Model Configurer	29
5.5	Step4: UML Validator	30
5.6	Step5: Selection of Multiple Values	30
5.7	Step6: Model Processor	31
	5.7.1 Creating ProcessingModel	31
	5.7.2 Convert ProcessingModel	31
	5.7.3 Analyze ProcessingModel	32
	5.7.4 Return Results of the ProcessingModel	33
5.8	Step7: Show Results	33
6	Further Work	34

1 Introduction

This document is meant to be a quick guide for making it easier to understand the *Performance Analysis*¹ module source code and how it works. This module was made by Borja Fernández, Isaac Trigo, Álvaro Iradier and Luis Carlos Gallego, under direction of José Merseguer, in the Centro Politécnico Superior, University of Zaragoza.

The *Performance Analysis* module works like a plug-in of ArgoUML application [2] and uses the GreatSPN tool [4] as analyzer. It is implemented in the programming language Java [5].

2 Getting started

In this section, it shows how can it runs the ArgoUML application with the *Performance Analysis* module.

There are two possibilities for using the ArgoUML: from CVS (Current Version Source) or from the binary distribution (latest stable release).

2.1 ArgoUML from CVS

ArgoUML is an open source project, so the CVS is the latest available version source. For this reason, it is possible that the version, that it is just downloaded, doesn't work. In order to develop with ArgoUML it is absolutely mandatory to get the CVS version of ArgoUML. Notice that the CVS contents is not only a set of source files but instead it is the complete development environment.

It must be said that some parts of the *Performance Analysis* module source code depend on the ArgoUML code (xmi conversion, for instance), so maybe there are versions of ArgoUML that won't work with the module and viceversa.

2.1.1 Getting ArgoUML

The first thing it will need to do, is to create a folder to download the ArgoUML files from the CVS must be created. For example: `~/argouml`. Once the folder is created, enter that folder and run:

```
cvs -d :pserver:anoncvs@cvs.tigris.org:/cvs login
```

That command needs to be run only the first time you're using the CVS. If prompted for a password, enter **guest**. Then, you can *checkout* the full CVS by running the following command:

```
cvs -d :pserver:anoncvs@cvs.tigris.org:/cvs checkout argouml
```

¹Last revised by Luis Carlos Gallego, February 25, 2005

This will download everything from the CVS, including website and documentation. If you don't want to get the full CVS, you can checkout individual modules, using:

```
cvs -d :pserver:anoncvs@cvs.tigris.org:/cvs checkout argouml/modulename
```

For building ArgoUML you need at least the following modules from the CVS:

- src_new/
- tools/
- libs/

2.1.2 Environment variables

For doing most building steps, the environment variable JAVA_HOME needs to be set correctly. For example:

```
JAVA_HOME=/usr/local/j2sdk1.4.2
```

2.1.3 Building ArgoUML

Once that the CVS repository is downloaded. From *argouml/src_new*, runs on of the following commands:

- **./build.sh** will compile ArgoUML sources.
- **./build.sh run** will compile ArgoUML and run it in the JVM.
- **./build.sh jar** will compile ArgoUML and build a .jar file in argouml/build folder.
- **./build.sh package** will compile ArgoUML and copy argouml.jar and all required .jar files in argouml/build folder

There are some other targets. For more info check ArgoUML documentation. The recommended target is *./build.sh package*. Then you can go to argouml/build folder and run ArgoUML with:

```
java -jar argouml.jar
```

2.1.4 Building the Performance Analysis module

In order to build the performance analysis modules, you need to have the ArgoUML sources from the CVS, and have it correctly built, so `argouml.jar` is in `argouml/build/` folder. You need JDK 1.3 or higher.

There are two possibilities to obtain the *Performance Analysis* source code: extracting/copying a zipped archive or getting the source code from CVS server. For that, you must previously register as user. Then, you have to do:

```
cvs -d :pserver:user@rdp99:/home/cvs/cvsroot login
```

Enter your password. If you want to change it, type:

```
cvs-passwd
```

Then,

```
cd destination_path
cvs -d :pserver:user@rdp99:/home/cvs/cvsroot checkout performanceanalysis
```

As result, you should have a tree similar to:

```
argouml/
  modules/
    performanceanalysis/
      build.xml
      deprecated/
      doc/
      grammar/
      lib/
      makeModules.sh
      module.properties
      src
      TXTs/
```

Then, from the `argouml/modules/performanceanalysis/` folder run the commands:

```
../../tools/ant-1.4.1/bin/ant clean
../../tools/ant-1.4.1/bin/ant install
```

First command is not really necessary, but cleaning the source tree before compiling it can save some trouble, so we strongly recommend it. Here we're using the *ant* tool included in the *tools/* module, from the ArgoUML CVS. Probably it's possible to use another external version of *ant*.

There are some other *ant* targets available, just take a look at `build.xml` to see which ones are available. The *install* target will build the module, create the `performanceanalysis.jar` file, and copy it to `argouml/build/ext` folder, so when you run the ArgoUML program in `argouml/build`, the performance analysis module will be found and loaded.

2.2 ArgoUML from binary distribution

2.2.1 Getting ArgoUML

Another way it is to use a binary distribution of ArgoUML. In this case, the first step is to download it from the ArgoUML web site: [2] and selects the latest stable version.

2.2.2 Running the Performance Analysis module

For running the *Performance Analysis* module, it must have a previous compiled (see Section ??) one and copies it into *argouml/ext/* directory.

Once this is done, to execute ArgoUML you can go to *argouml/build* folder and type:

```
java -jar argouml.jar
```

3 Source code structure

3.1 Overview

The tool follows the architecture proposed in the *UML Profile for Schedulability, Performance and Time Specification* [6, Cap.9], that establishes a components division based on its functionality: the *Model Editor*, the *Model Configurer* and the *Model Processor*.

In the current implementation of the module the *Model Editor* function is made with ArgoUML. We hope to implement that function with Rational Rose too. The *Performance Analysis* module is inserted in ArgoUML like a plug-in. That plug-in is showed in the ArgoUML interface as a submenu in the tools menu of the menubar and makes the *Model Configurer* and the *Model Processor* (the *Model Converter*, the *Model Analyzer* and the *Results Converter* functions). So, the process starts with a model in ArgoUML, then that model is exported to XMI format. If it's needed then a configuration of the parameterized model is performed with a configuration archive and then the new configured XMI file is converted to a new stochastic Petri Nets model. This last model is sent to the *Model Analyzer*. The *Model Analyzer* functions are now made by the GreatSPN tool [4]. This one returns the results of the analysis to the plug-in, the *Performance Analysis* module is in charged of processing and showing in a coherent way in the *Model Editor*.

But there is another main function that our module does and it's not commented in the *UML Profile for Schedulability, Performance and Time Specification* [6, Cap.9]. Due to the fact that some UML models and diagrams were not valid some execution errors were raised. These errors make the developing process of the module harder because we didn't know if they were programming errors or user errors. So a *UML Validation Module* has been developed in order to check the models before calling the *Model Processor*. This part is performed after configuring the model.

3.2 Scalability

The main idea of the structure is to put on an equal footing with the proposed profile [6]. So reason we must put into different packages every module of this mentioned profile. And we want to give the module the more functionality without having the need of changing everything that is implemented yet.

So, each package have to make the most generally that it is possible, for this reason we must make abstract classes of every package. Then, we make particular classes based on these abstract that implement the code. In this way in a further work it will be able to change any package or component into another that satisfies the abstract class: using Rational Rose [1] instead of ArgoUML in the Model Editor, using PNML [7] as the format to store the Petri Net, instead of the GSPN format or another Petri Nets analyzer in the Model Processor parts (Model Convertor, Model Analyzer and Result Convertor).

For obtaining this flexibility we use interfaces, the factory pattern and the singleton pattern. Following we explain this concept with a real example implemented in the module. This example explains how two different *ModelEditor* can be used in the model changing only one thing in the whole module source code.

We define an interface that defines the methods that must have every *ModelEditor*. Then, if we want to use Rational Rose as Model Editor, this interface is implemented by the *ConcreteXMIRoseModelEditor*, that will generate XMI files in Rational Rose format. The only condition is that it must return a *Model* class. Suppose now that we have this *ConcreteXMIRoseModelEditor* and another one that will generate, via ArgoUML, those XMI files and that is called *ConcreteXMIArgoUMLModelEditor*. Now to choose one or another in the module we use the factory pattern. We create an abstract class of the factory, *ModelEditorFactory*, with the methods that that factory will export. In our example one method: *createModelEditor*. We implement two factories that will provide any part of the module with a concrete *ModelEditor*, one will return a *ConcreteXMIRoseModelEditor*, and the other a *ConcreteXMIArgoUMLModelEditor*. So a call to a method called *createModelEditor* is made in any part of the module that needs that *ModelEditor* and a concrete *ModelEditor* will be returned. The factory will have to be only one, so we use the singleton pattern to get this. That will allow us to have only one instance of the factory in the whole model. We have said that the *ModelEditorFactory* is an abstract class, because it implements the method that will return the single instance of the factory. Just modifying that method, and selecting which concrete factory is returned will make the whole module use a *ConcreteXMIRoseModelEditor* or a *ConcreteXMIArgoUMLModelEditor*. That's why this module is highly scalable.

So, the complete implementation in the module of the *ModelEditor*, following the singleton and factory pattern and using interfaces and abstract classes for a higher scalability is the following:

- *ModelEditor*: Generic interface for any *ModelEditor* containing its methods. In our case generateXMIModel.

- *ModelEditorFactory*: returns the *ModelEditor*. This is implemented as Singleton pattern, for this reason there is always an instance at the most. This is implemented as an abstract class that creates the concrete factory, and defines the method all *ModelEditorFactory* should have, createModelEditor.
- *ConcreteModelEditorFactory*: This is the extension of the *ModelEditorFactory*. It implements the createModelEditor method and returns an *XXXModelEditor* where XXX is, for instance, ArgoUMLXMI, or something like this. That means that is a Model Editor that is plug-in in ArgoUML and that returns UML models in XMI format.
- *XXXModelEditor*: another interface just in case some new methods a concrete instance of the *XXXModelEditor* should have. It's an extension of the *ModelEditor*.
- *ConcreteXXXModelEditor*: it's the implementation of the *XXXModelEditor* and *ModelEditor*. It's the part that really implements the Model Editor function.

3.3 Directories structure

Here we present the structure of the source code and a comment about what we will find in those directories:

- **edu.cps.performance**: This package contains the abstractions definitions for the Model Editor, Model Configurer, Model Processor and UMLValidator, as commented in the previous section. The above abstractions are implemented by the *Factory* pattern and contain the specifications of the basic functions that any concrete component must have. So, we find here the interface of these parts and the abstract classes of its factories.
- **edu.cps.performance.error**: This package includes the definitions of exception classes that the tool can raise. Mainly, there are two errors types: User Errors and and Program Errors.
- **edu.cps.performance.utils**: Auxiliary classes like option dialog, debug window, debug messages, configuration keys, logger information, multiple value selection windows, storage of previous results or checking functions.
- **edu.cps.performance.model**: This package contains the work flow control of the entire module. Its main class is *Model*. This class stores every data that will be needed in any part of the module. This includes XMI files, configuration tables, parsed XMI models, results etc. We find here another class called *Tool*, implemented following the *Factory* pattern. So, we find *Tool* as the interface, *ToolFactory* as the abstract class that will return the concrete *Tool* and *DefaultTool*, that implements most of the methods of the interface. In this particular case there are a couple of

methods that depends on the *ConcreteXXXModelEditor* used, so they will be extended later. The purpose of this *Tool* is to handle all the XMI tag, and make it easier to find things in the XMI models. For instance, if we are looking for a node in the XMI models we are searching for a tag like this: *Foundation.Core.Node*. The *Tool* class defines all these tags so that in the module we can look after them in an easier way.

The subdirectories in this directory are the following:

- **edu.cps.performance.model.xmi**: it contains all the classes in charge of dealing with the XMI files containing the model. We can find classes for traversing the XMI files, configuring or parsing them, transating, generating or handling them. All these classes can only be accessed via the *XMIFactory* class that is found here too. We found here the *UMLConverter* but this class should be in the *ModelConverter* because it defines the methods that any part of the module that wants to translate an UML model into anything (Petri Nets for example) must have. So, although it deals with the model we would move it to the *ModelConverter*. It's not moved yet because that part of the module is the most crossed of the whole module, but this will be discussed later. The classes found here are *XMITranslator*, that is used to translate the elements in the XMI model. The *XMITraverser* that is used to traverse the parsed XMI file and find all information needed. The *XMIParser*, that parses the XMI file and stores it in a DOM tree, making a better access to the UML models. The *XMIHelper*, that is used to find some elements in the XMI files, such as names of the element, tags and so on. *XMIGenerator*, that is the interface that any generator of XMI must have. It must be implemented with the corresponding code used by the concrete tool, in our case ArgoUML. *XMIConfigurer*, it configures the XMI model if it's needed.
- **edu.cps.performance.model.elements**: it stores elements of UML models, so that we would be able, if we want, to use them instead of the XMI files. Now they are used with those files , because we do not store the complete UML model, but the states, transitions, events and so on, selected by the user to make an specific query. Then, the classes that are defined here are
- **edu.cps.performance.model.results**: Class that stores all the results that can occur during the performance analysis.
- **edu.cps.performance.model.parsedxmi**: Classes that will store the parsed XMI file, and the dictionary.
- **edu.cps.performance.model.editor**: It contains the information referred to the *Model Editor* in the profile. So depending on the CASE tool that we will use, we will make a package with all necessary items to use it. In our case the subpackage is:

- **edu.cps.performance.modeleditor.argouml:** for the specific CASE tool (ArgoUML). There are two different parts in this sub-package. The one handling with the XMI generation of the model and the one dealing with the concrete CASE tool, in our case ArgoUML.
 - * **edu.cps.performance.modeleditor.argouml.connection:** it plugs the *Performance Analysis* module to the CASE tool. This package is not possible to generalize, because it is specific to the chosen CASE tool (in this case ArgoUML). The specific way to connect our tool to ArgoUML will be discussed later. So, the classes in this package will be discussed there.
 - **edu.cps.performance.modeleditor.argouml.connection.actions:** Here we find the actions that are triggered when some option in our menu in the tool is chosen.
 - * **edu.cps.performance.modeleditor.argouml.xmi:** This part is in charged of generating the XMI model using ArgoUML to achieve this. So, we find here the XXXModelEditorFactory, the XXXModelEditor, and the ConcreteXXXModelEditor, in this case, *ArgoUMLXMIModelEditorFactory*, *ArgoUMLXMIModelEditor* and *ConcreteArgoUMLXMIModelEditor*. This is the implementation explained in the subsection Scalability. But there are also three extensions of three classes, the ArgoUMLXMITool, that implements the methods that the DefaultTool couldn't, the ArgoUMLXMIHelper that extends the XMIHelper, class that was in edu.cps.performance.model.xmi as well as the XMIGenerator that is also implemented here in the ArgoUMLXMIGenerator. This extensions are needed because the generation and handling of the XMI files are dependant on the way that ArgoUML generates them and writes them. It must be said that, whenever a change in the storing of the UML diagrams is made in the ArgoUML source code, this ArgoUMLXMIGenerator must have been changed. I strongly recommend to study the export to XMI plug-in that comes with the ArgoUML source code every time a new stable or instable ArgoUML is released and we want to use it. If this export plug-in is different from the ArgoUMLXMIGenerator method to generate the XMI this method must be changed for the one used in the plug-in.
- **edu.cps.performance.modeleditor.rose:** in a further work, same idea that the argouml package.
- **edu.cps.performance.modelconfigurer:** This packages includes the items regarding concrete model configuration. In other words, we have the generic class for the Model Configurer and here we extend it, it's the same idea that is commented in the Model Editor package. In our case, we

configure XMI files. It's needed to say that the configuration is not read later in the process from the XMI file, that will be too slow. It's stored in a class here called `AnnotationTable`, and all the annotations, the ones that needed configuration and the ones that didn't, are stored here and will be consult here.

There are two subpackages here that defines the TVL grammar and a subgrammar of this one, but with new methods, that we have called GRM grammar.

- **edu.cps.performance.modelconfigurer.tvlgrammar:** Classes generated by javacc that defines the TVL grammar that will be discussed later. There are also here classes dealing with the perl interpreter that will read the values and help in the configuration of the model. Now the part of the tvl grammar here is deprecated and it's used the one below, but the part of the perl interpreter is still used.
 - **edu.cps.performance.modelconfigurer.completetvlgrammar:** Defines and extends the same grammar but can store multiple values. So this grammar is more complete and it's used instead of the previous one, because this ones involves the other and extends it.
 - **edu.cps.performance.modelconfigurer.grmgrammar:** A grammar similar to the tvl one, but that will be used in the module to note which classes are in which nodes, so distinct methods are implemented here.
- **edu.cps.performance.validation:** we find here the classes that are used to check if a UML model is valid. We say that a model is valid if it can be converted to Petri Nets and all the annotations in the model could be interpreted by the processor. This model is implemented to avoid error messages with no explanation during the processing part of the module. So, we find here the `concreteValidator` class that implements the `UMLValidator` explained in the `edu.cps.performance` part. This class is also implemented with its factory and its interface. The class that stores all the warnings and errors that the UML model has can be found here too.
 - **edu.cps.performance.modelprocessor:** Here we find an extension of the `ModelProcessor` base class to Petri Nets with its following factory pattern. Same idea that was follow and explained in the `ModelEditor` package. Following the profile, this part is divided in `Model Convertor`, `Model Analyzer` and `Results Convertor`. So, here we also can find the abstraction definitions of the `ModelConvertor`, `ModelAnalyzer` and `ResultsConvertor` with its respective abstract classes factories. It's the same schema used in the `edu.cps.performance` package with the `Model Editor`, `Model Configurer`, `Model Processor` and `UMLValidator` parts.

Then, the package hierarchy is the following:

- **edu.cps.performance.modelprocessor.model:** equivalent to edu.cps.performance.model. Here we store all that is needed to transform the UML model and analyze it. The main class is ProcessingModel and the idea is similar to the one that led us to create the class Model previously commented. Here we will store the parsed configured XMI model, with its annotation table, the results of the conversion and the results of the analysis. Due to the fact that we're dealing here with Petri Nets in GSPN format a subpackage is created, equivalent to the one created in edu.cps.performance.model to handle xmi:
 - * **edu.cps.performance.model.greatspn:** here we store a GSPNFactory that will provide GSPNHelper, that will handle the use of the files with the stored Petri Nets. It's also in charge of analyzing those files, merge them and so on. There are also stored here the classes that store information of the GSPN Petri Nets: GSPNTransition, GSPNPlace or GSPNPetriNet.
- **edu.cps.performance.modelprocessor.modelconverter:** here we find no class, but subpackages. If some more abstractions of the Model Converter arise then they should be put here. The subpackages found define the concrete Model Converter used, that will translate the UML models into, Petri Nets (or whatever, Markov chains etc) in a concrete format. The format used now is the one that GreatSPN used, but we could also use pnml or whatever.
 - * **edu.cps.performance.modelconverter.greatspnconverter:** In this subpackage we found the implementation of the concrete Model Converter and Model Converter Factory, the same way that it was implemented with the Model Editor. So we find here the ConcreteGreatSPNModelConverter that translates an UML model into Stochastic Petri Nets in GreatSPN file format. It uses the class PNTranslator (that uses the factory pattern too), and is the one that really makes the translation. But the PNTranslator does not merge the Petri Nets results of the translation into one Petri Net of the whole model. The ConcreteGreatSPNModelConverter analyzes what the user wanted to know and merge the Petri Nets if it's needed, or leaves it without merging. This is implemented this way because there are certain queries such as Transmission Speed, Message Delay, that do not need one merged Petri Net. This package contains code from the project that started this module, when the profile didn't exist, so the subpackages and so on are not very well ordered.
 - **edu.cps.performance.modelconverter.greatspnconverter.errors:** This part contains extensions of the errors defined in edu.cps.performance.error that can happen only during the conversion process.

- **edu.cps.performance.modelconverter.greatspnconverter.generation:**
This part contains all the classes that transforms an UML model into Petri Nets. All extends the class UMLConverter described in edu.cps.performance.model.xmi. There one abstract class, PetriNetBuilder, that implements the methods to write the Petri Nets in the GSPN files, and defines methods for translation. The rest of the classes extends this one and translates the different diagrams into petri nets, the class diagrams, the state diagrams, deployment diagrams and activity diagrams. These classes are SDtoGSPN, ADtoGSPN, SMtoGSPN, that are quite similar. ClassInfoGetter that recovers information from the Class diagram and DeploymentInfoGetter and DeploymentDiagramReader that recovers information from the Deployment Diagram.
- **edu.cps.performance.modelconverter.greatspnconverter.results:**
Stores data produced during the conversion process. The main class here is ComponentPetriNets. This name is not accurate because not only the Petri Nets of every component of the model is stored there, but all the model info, including transmission costs calculated between nodes, all the classes info, name, population and so on.
- **edu.cps.performance.modelconverter.greatspnconverter.storage:**
Here we can find many things, not well ordered, but we couldn't find any better organization. We find classes that stores info about the UML models, that will be used in the translation such as Actions, Activities etc. There are also here classes that will store information of the translated Petri Nets, such as GSPNEstados or GSPNTransiciones. All the classes named in Spanish corresponds to the first project, and they all are used in the translation, but a reconfiguration of these classes will be helpful to understand the performance of the module. We find in here too a subpackage with the most important information in the translation. It should be in a more important place, but no other better place in this conversion part of the module has been found:
- **edu.cps.performance.modelconverter.greatspnconverter.storage.modelinfo:**
It stores all the info in the UML diagrams. Classes info, Event info, Physical Nodes info, Communication info, activities info and so on. This is the info that will be used later to translate the model into petri nets.

A complete description of the conversion process will be commented later.

- * **edu.cps.performance.modelconverter.pnml:** this subpackage does not exist, but it's commented here as a possible

extension to this module. This package would have a ConcretePNMLModelConvertor, that would transform the UML model into Petri Nets in PNML formal. It must be said that we must not reimplement the whole conversion, because the storage, results etc parts can be used again. In fact, the only class that will need to be implemented is the one will write into files the Petri Net, the *PetriNetBuilder* abstract class commented before in the edu.cps.performance.modelconvertor.greatspnconvertor.generation part.

- **edu.cps.performance.modelprocessor.modelanalyzer:** This part deals with the analysis of the Petri Nets to ask the query the user has made. The queries the user can make are explained in section XXX. This package contains no classes, as the edu.cps.performance.modelconvertor one, but a subpackage containing the classes to make the analysis using the GreatSPN tool.
 - * **edu.cps.performance.modelanalyzer.modelprocessor.greatspnanalyzer:** We find here, following always the same pattern used until now, the concrete Model Analyzer class. This time this analyzer uses the GreatSPN tool.
 - * **edu.cps.performance.modelprocessor.modelanalyzer.anotherTool:** This package does not exist, but it's here to illustrate how another tool, not the GreatSPN one, can be used here instead of that one.
- **edu.cps.performance.modelprocessor.resultconvertor:** Following the same idea used in edu.cps.performance.modelprocessor.modelanalyzer and edu.cps.performance.modelprocessor.modelconvertor, we find here a subpackage in charge of returning the results in a way that the tool used as Model Editor can understand.
 - * **edu.cps.performance.modelanalyzer.resultconvertor.argouml:** Classes that implements the Result Convertor, following the same pattern used until now. Here we find the concrete result convertor, that will translate the results that the analyzer got (GreatSPN in our case) into something that the tool used to get the UML models can understand (in our case ArgoUML).
 - * **edu.cps.performance.modelprocessor.resultconvertor.rose:** Once again this package does not exist, but it's here to illustrate where to place the classes to convert the results for another tool, such as rational rose.

Apart from these proper packages of *Performance Analysis* source code, it is necessary to use external packages, like *xerces* to parse XML and the ArgoUML API library to generate XMI, as it was commented before .

4 Components description

As it was said before the architecture for the Model Processing tries to follow the proposed one in the *UML Profile for Schedulability, Performance and Time* [6].

Here we'll comment how the Model Editor, Model Configurer, UML Validator and Model Processor work. The last one makes the real analysis tasks. It is composed by the *Model Converter*, the *Model Analyzer* and the *Result Converter* that will be commented too .

4.1 Model Editor

As it has just mentioned in previous sections, the Model Editor functionality [6, Cap9] is realized via the ArgoUML application. ArgoUML is a CASE tool to allow to work with UML models (use case diagrams, class diagrams, etc.).

Once that the required model is correctly made from ArgoUML (the Model Editor), the next step is to analyze its performances. Therefore, as result of this phase, it obtains an XMI model containing the UML diagrams modelled in the ArgoUML tool.

The Model Editor package that we have developed have two main parts. The first one deals with the integration of the whole module in the ArgoUML tool, and the second ones performs the Model Editor function properly said, as it's presented in the profile. This second part deals then with the generation of the XMI model. Now we're going to present this two parts in a more extensive way.

4.1.1 Plugging-in to ArgoUML

To make our module work, we first needed to plug it into one case tool. We chose ArgoUML. To achieve the connection to ArgoUML we must create a class that extends the JMenu, to insert it as a Menu in the ArgoUML tool. It must implements the PluggableMenu interface defined in the ArgoUML source code. We call this class *PerformanceAnalysis* and is found in edu.cps.performance.modeeditor.argouml.connection. Once it's implemented we insert some menu options in that class and attach them to an action that we must develop and that has to implement another interface defined by the ArgoUML developers. This is our *PAModuleAction* that implements UMLAction. This class is the one that calls all the parts in the module, the generation of the XMI in the Model Editor, the configuration of the model, the processing and finally it shows the results. We will follow a complete execution of the module later, from the moment the menu is clicked until the results are showed.

To get more information about how to make a plug-in into ArgoUML go to the ArgoUML site (INSERT CITE) or read Isaac Trigo's project.

4.1.2 Generation of the XMI File

This is the main function of the model editor, to generate an XMI file containing all the information of the UML models. It's performed by the concrete

Model Editor class that calls the Concrete XMI Generator. This generation has changed several times, every time that the ArgoUML developers change the classes that they use to store the UML model. So every time a new ArgoUML version is released this part must be checked. We recommend to check the ArgoUML plug-in to export to xmi and copy it where it must be in the ArgoUMLXMIGenerator class.

Once the model is saved as XMI file it's stored in the class *edu.cps.performance.model.Model*, and this class will be passed to the Model Configurer.

4.2 The Model Configurer

Once we have an XMI file containing the UML model it must be checked to find out if the model is parameterized and configuration is needed. The Model Configurer [6, Cap9] functionality is to convert a parameterized UML model into a concrete model. Apart from that main functionality our Model Configurer will store all the annotations in the model in an annotation table, to make the access to these annotations easier and faster. An annotation is a set of stereotype, tag and values. This annotations are inserted in the UML diagrams as comments or tag-values. They must follow the TVL grammar that will be discussed later, or the GRM one if the stereotype is GRMcode. We will explain this in the following sections.

XMI parameterized model configuration is described as steps 2 and 3 (Model Configurer) in [6, Page 9-2]. Target of these steps is to replace the existing tagged values with parameterized or valuable expressions, written in TVL (Tagged Value Language, which is a subset of *Perl*, see [6, Ap-A]), with the equivalent evaluated expressions.

Performance Analysis annotations should be done, as described in [6] and [3], in elements extended with one of the stereotypes defined in [6], and specified as tagged values. However, we support an alternative notation. Performance annotations can be done in *Comments (Notes)* using the following syntax:

```
«Stereotype» {tag1=value1.1;value1.2;...;valuen,
tag2=value2.1;value2.2;...;value2.m,...,tagy=valuey.1;...}
```

where *tagX* is a tagged value tag, and *valueX.n* is the *n* possible value for that tag. As it's showed a tag can have multiple values. The configuration process is done by the *edu.cps.performance.modelConfigurer.ConcreteXMIModelConfigurer*, using the *edu.cps.performance.model.xmi.XMIConfigurer*. The concrete Model Configurer checks if the model has been configured before. If so it returns the annotationTable with all the previous annotations. If it hasn't been configured before the the model is configured. To configure the model this *ConcreteXMIModelConfigurer* calls the *XMIConfigurer*. But before calling the concrete model configurer parses the XMI file containing the model. Then the parsed xmi file, stored now in as a DOM tree, is passed to the XMIConfigurer.

This one configures the model in two phases. These two phases are the following:

1. **First phase, traversing of the DOM tree:** the DOM tree is traversed from the root node, and recursively into children nodes. Two types of elements (Comments and Stereotypes) are processed, while other nodes are just ignored (but recursively traversed). The configuration process starts by calling the *XMIConfigurer.configure(Node root)* method. This method calls the *auxTraverse(node)* method, which recursively searches in the tree.

When a *Comment* node is found (this is the XMI name of the UML *Notes*), the *parseComment* method is invoked. There, the comment is parsed using the TVL grammar (see below), and valuable subexpressions are found and evaluated (evaluated (see below)). Then, if the comment is valid for our *Performance Analysis*, the parameterized comment node on the DOM tree is replaced with the evaluated equivalent expression.

If a *Stereotype* node is found, it calls *parseStereotype*. This method extracts a list of XMI IDs referencing the model elements that are extended with this stereotype. These element IDs are stored in a list, which will be used in second phase of the configuration.

2. **Second phase, configuration of Stereotypes:** we search in the DOM tree for every of the Elements extended by known Stereotypes (the XMI IDs were stored in a list in first phase) using the method *XMIConfigurer.searchByID(root)*. Then each element is passed to *parseTaggedValues(element)*. This method extracts the values from every tagged value in the stereotyped element, and evaluates the expressions, replacing the old values with the evaluated ones in the DOM tree.²

Once the two phases are completed, the in-memory DOM tree is fully configured, and no parameters are missing. The document is then serialized to a XMI file, using *writeConfiguredToXMI(p.doc())* method in the concrete XMI-Generator, *ArgoUMLXMIGenerator*. But during the configuration process not only the now configured annotations are stored, but all the other annotations that weren't parametrized. They are all stored in an *AnnotationTable* class. The configured XMI file will be saved in the same temporary directory as the parameterized XMI model. It must be said that, although the XMI model is saved it won't be used, because we have parsed the XMI file in the DOM tree which provides faster access to the model components and the *AnnotationTable* containing all the annotations of the model. So we will use this DOM tree and *AnnotationTable* instead of dealing with the configured XMI file.

These DOM tree, *AnnotationTable* and configured XMI file will be stored in the *edu.cps.performance.model.Model* class. This class will be passed now to the UML Validator part.

²Right now, we evaluate all tagged values, just ignoring the tag name. This can be improved so only known tags (for the element stereotype) are processed, ignoring other tagged values. NOTE: this problem is solved because the UML Validator will check the tags.

For more details about TVL grammar and how it works in the *Performance Analysis* module, see the Section 4.2.1.

4.2.1 The Tag Value Language

TVL is a simple subset of Perl, the evaluation of TVL expressions is done by an external Perl interpreter (which can be configured in the module Options dialog).

The Perl interpreter is loaded with a small Perl program that enters an evaluation loop. The evaluation loop will take one line from the standard input, and output the result of evaluating the input (followed by a mark line, just for synchronization purposes). The Perl interpreter process is launched when the configuration step begins, and standard input, output, and error output are piped into the module, so expressions are written to the process input pipe, and results(errors) are read from output pipe(error pipe). Note that TVL expressions can contain variables, such as *\$value*. Before telling Perl to evaluate an expression, a search for variables in the expression is done. When a variable is found for the first time, the user is prompted to choose a configuration file. This file will be evaluated by the Perl interpreter line by line, and then the normal configuration process will continue. So, if the file contains a line like:

```
$value=5;
```

Then variable *\$value* will be defined, and replaced by the Perl interpreter when found inside an expression. Configuration files are written in Perl, and can have any legal Perl expressions, or comments. Each command must be finished with ";". For example:

```
#A Performance Analysis configuration file
#written in perl
$value=5;
$value2=10;
#Here, a perl expression
$value3=$value*$value2;
#A conditional expression.
#If $value3<40, $other will be assigned 100-$value3
#If $value3>=40, $other will be assigned 100
$other=($value3<40)?100-$value3:100;
```

The TVL grammar specification The grammar used for parsing Comments and Tagged Values is defined in file *TVLGram.jj*, included in the source code, *grammar/PAGramMultiple* folder. In the *grammar/* folder we find another couple of directories *PAGram/* and *GRMGram/*. *GRMGram/* will be presented later, and *PAGram/* contains a simple version of the *TVLGram.jj* file, that only allowed single values. The current version of the module support multiple values in the expressions, so the version stored in the *PAGram/* folder is deprecated.

This grammar definition is processed by *JavaCC* [5], and some classes are generated. These classes are copied into the module package *edu.cps.performance.modelconfigurer.completetvlgrammar*. The generated parser includes some methods to parse an expression from a string (instead of default standard input stream), and to get the valuable subexpressions found in the TVL expression. Grammar for comments is described (as a summary) here:

```

TVLEXPRESSION: "<" STEREOTYPE ">" "{" PAEXPR "}"
PAEXPR: TAGGEDVALUE ("," TAGGEDVALUE)*
TAGGEDVALUE: TAG "=" VALUE ";" (VALUE)*
VALUE: LIST | DIRECTVALUE
LIST: "(" VALUE ("," VALUE)+ ")"
DIRECTVALUE: "'" QUOTEDEXPRESSION "'" | RAWEXPRESSION
RAWEXPRESSION:
    "(" RAWEXPRESSION ")" RAWEXPRESSION |
    "{" RAWEXPRESSION "}" RAWEXPRESSION |
    ["(" ")" "{" "}"] RAWEXPRESSION |
    epsilon

```

Grammar for *Tagged Values* values is the one starting in the VALUE production (as Comments notation is meant to be an alternative to Tagged Values).

How does the grammar parser works As it was said, the XMICConfigurer really does two functions. First, it configures the parameterized model by evaluating parameterized expressions using the Perl interpreter. Second, it adds configured annotations to an *AnnotationTable* class, which will be used later in the model translation to get the performance annotations easily, no matter they were as tagged values or in Notes.

There are two main methods in the class *PAExpr*, generated by javacc based on the file *TVLGrammar.jj* we wrote:

- *parseComment(String expression)* will parse the given expression as if it was a Note (alias comment) annotation. Notes usually look like:

```

<PAstereotype> {tag_value1=setOfexpressions1,
tag_value2=setOfexpressions2,...}

```

The parser will extract the following information: **stereotype**, **tags** (first and last index in the string for each tag value), and **multiple valuable subexpressions** (first and last index for each subexpression inside the expressions, and first and last index of the each value. Note that expressions can be lists, so there can be subexpressions inside a tag value expression).

- *parseValue(String expression, String stereotype, String tag)* will parse the given expression as if it was a tagged value (only the multiple values, not the tag). "*expression1*" or "*expression2*" in the previous Note example are tagged values. In this case, there's no stereotype or tag name definition,

so they must be given (it's not really necessary, but it's recommended, for managing Tag values and Notes in a similar way).

Both methods will return 0 if parsing was done correctly, and <0 if there was an error. After having parsed an expression correctly, information about the parsing can be retrieved using the following methods:

- *getTVLEExpr()* will return the parsed. This string will be similar to the original expression, but any blanks will be removed. Any indexes returned by the following functions are referred to character positions in this string.
- *getStereotype()* will return the stereotype of the last parsed expression.
- *getSubExprCount()* will return the number of valuable subexpressions found.
- *getTagCount()* will return the number of tag names found in the expression (only one if *parseValue* was called).
- *getValuesCount()* will return the number of values the set of expressions has. This method is used for multiple values support.
- *getSubExprBegin(int n)* will return the index (inside the parsed expression string) where the n-th valuable subexpression begins.
- *getSubExprEnd(int n)* will return the index (inside the parsed expression string) where the n-th valuable subexpression ends.
- *getTagBegin(int n)* will return the index (inside the parsed expression string) where the value for the n-th tag in the expression begins.
- *getValueBegin(int n)* will return the index inside the parsed expression string) where the n-th value in the set of expression begins.
- *getTagEnd(int n)* will return the index (inside the parsed expression string) where the value for the n-th tag in the expression ends.
- *getValueEnd(int n)* will return the index inside the parsed expression string) where the n-th value in the set of expression ends.
- *getTagName(int n)* will return the name of the n-th tag in the expression.

Two examples to make it clear. We call

```
PAExpr.parseComment("«PAtest»{patag1=5,patag2=($time,'s')}")
```

and then we try:

- *PAExpr.getTVLEExpr()* → "«PAtest»{patag1=5,patag2=(\$time,'s')}"
- *PAExpr.getStereotype()* → "PAtest"

- *PAExpr.getSubExprCount()* → "3"
- *PAExpr.getSubExprBegin(0)* → 19
- *PAExpr.getSubExprEnd(0)* → 20
- *PAExpr.getTVLExpr().substring(19,20)* → "5"
- *PAExpr.getSubExprBegin(1)* → 29
- *PAExpr.getSubExprEnd(1)* → 34
- *PAExpr.getTVLExpr().substring(29,34)* → "\$time"
- *PAExpr.getSubExprBegin(2)* → 35
- *PAExpr.getSubExprEnd(2)* → 38
- *PAExpr.getTVLExpr().substring(35,38)* → "'s'"
- *PAExpr.getTagCount()* → 2
- *PAExpr.getTagName(0)* → "patag1"
- *PAExpr.getTagName(1)* → "patag2"
- *PAExpr.getTag().substring(PAExpr.getTagBegin(0),PAExpr.getTagEnd(0))*
→ "5"
- *PAExpr.getTag().substring(PAExpr.getTagBegin(1),PAExpr.getTagEnd(1))*
→ "(\$time,'s')"

Now we use

```
PAExpr.parseValue("$USERS1+1;2","PAothertest","PAusers")
```

the trace is:

- *PAExpr.getTVLExpr()* → "\$USERS1+1;1"
- *PAExpr.getStereotype()* → "PAothertest"
- *PAExpr.getSubExprCount()* → "1"
- *PAExpr.getSubExprBegin(0)* → 0
- *PAExpr.getSubExprEnd(0)* → 11
- *PAExpr.getValueCount()* → "2"
- *PAExpr.getValueBegin(0)* → "0"
- *PAExpr.getValueEnd(0)* → "9"
- *PAExpr.getValueBegin(1)* → "10"

- *PAExpr.getValueEnd(1)* → "11"
- *PAExpr.getTVLEExpr().substring(0,9)* → "\$USERS1+1"
- *PAExpr.getTagCount()* → 1
- *PAExpr.getTagName(0)* → "PAusers"
- *PAExpr.getTag().substring(PAExpr.getTagBegin(0),PAExpr.getTagEnd(0))*
→ "\$USERS1+1;2"

4.2.2 Configuration and model annotation

Now, using the methods explained in the previous subsection, show how is done model configuration.

When a Note is found, we call the *PAExpr.parseComment(comment)* method. If no error is given, we call the *XMICompiler.evaluateCompleteTVLEExpression()* method, which will return the evaluated comment, and then we replace the *CharacterData* in the Comment node with the evaluated comment.

If a tagged value is found, *PAExpr.parseValue(tagvalue, stereotype, tagname)* is called. If there are no errors, we call *XMICompiler.evaluateCompleteTVLEExpression()* method, which will return the evaluated tag value, and we replace the *CharacterData* in the Tag Value node with the evaluated tag value, similar as done with Notes.

The *evaluateCompleteTVLEExpression* method simply needs to get all valuable subexpressions (calling *PAExpr.getSubExprCount()*), pass them to the Perl interpreter, and return a new string where these subexpressions have been replaced by the output of the Perl interpreter. This method simply ignores if the expression was from a Note or from a tagged value, but it doesn't matter. And does so for each value.

At same time, for every tag in the evaluated expression, the method adds an entry in the Annotations Table (*pannotations* variable, of class *AnnotationTable*), in the form: ELEMENTID: «Stereotype» tag=value. This provides a way for the model convertor to get the performance annotations in a simple way (it just needs to provide the analyzed element *xmi.id* and the desired stereotype and tag name) with no need to recursively search in both tagged values and notes.

In the end of this process we will have an annotation table storing every stereotype, tag and value in the model. It must be noticed that the same stereotype and tag for the same element may have several values, so methods to handle the annotations are implemented in the *AnnotationTable*.

4.2.3 The GRM Grammar

An special case to all the annotations are the ones that indicate which classes reside in which nodes. This annotations are only stored in the form of a comment in the Physical Node in the Deployment Diagram. To handel this annotations

we could have used the the TVL grammar, but there are two reasons not to use it:

- We thought that the GRMcode hasn't tag, we were wrong, but by the time we realized this the implementation was done.
- Second, an more important, new methods to handle the items listed in the grammar are used.

So, it's implemented the *GRMGram.jj* file in the *grammar/GRMGram* folder that was presented above. This grammar is quite similar to the TVL one, but only accepts a tag with several identifiers, that will be the classes resident in the node. When the grammar and the methods are created javacc is called and the classes in *edu.cps.performance.modelconfigurer.grmgrammar* are created. This classes provides same methods to parseComment and so on that the TVL grammar classes provided, and also methods to get the classes resident in the model.

In the end the annotations about the classes resident in the node will be stored in the AnnotationTable, but will have to have an special treatment, because, if several classes reside in a node, the stereotype GRMcode with the tag GRMmapping of the element node will have multiple values. But there are not multiple values, but the classes resident in the model, so methods to handle this are implemented too in the AnnotationTable, that will return all the classes that reside in the node, and that will not make the user choose which multiple values to use (this user decision will be presented later).

4.3 UML Validation

Once the model has been configured we check if the model can be processed without raising user errors. This module was implemented because when we were probing the PAModule several error arise with no comments, and we realized that they all were errors that the user made creating the model. So, we decided to check the UML models in order to be able to say the user that his model wasn't correct, and not to process the model when it wasn't going to end.

We check the model at this time because we have now all the information we need, we have all the configured model parsed and all the annotations stored in the AnnotationTable.

The validation process is made traversing the DOM tree and checking each element that is find during the traversing. For example, if we find a class, we will check that its annotations ar correct an so on.

This UML Validation part will raise errors if the model cannot be processed, and informs the user of that. But it also can raise warnings, that tells the user that the model can be processed without raising errors, but the results may not be what the user wanted.

The basic things that are checked in the UML models are (summarized):

- *Stereotype and tags:*

- PArepsTime, PAprob, PAsize and PAspeed tags only have sense if its stereotype is PAstep. PAinitialState tag is part of the stereotype PAinitialCondition. PApopulation is part of the stereotype PAcloseLoad. GRMmapping value must be of GRMcode.
 - The value of the PArespTime is a duration one, for example, only a number will raise a warning because it hasn't got units. A correct value that won't raise an error nor a warning is, for instance, (15,'s'). Valid units are ms, s, m, h and so on.
 - the values of a PAprob tag must be real values between 0.0 and 1.0
 - PAinitialState valid values are \$true or \$false.
 - PApopulation values must be integers.
 - PAspeed value must be similar to the PArespTime one but with speed units, such as Kbps, KBps, Mbps, MBps and so on.
 - PAsize value must be similar to the PArespTime one but with size units, such as bs, MBs, Mbs, KBs and so on.
 - GRMmapping values must be identifiers of existing classes.
- *State Diagrams:*
 - PArespTime is a tag that must be in an state with a do-Activity.
 - Every do-Activity must have a name. This restriction is needed because ArgoUML give us no other alternative to find a do-Activity but the name. So it's not a restriction imposed by the UML models.
 - PAprob is a tag that must be in a transition.
 - The sum of all probabilities of a set of transitions whose source is the same must be 1.0.
 - PAprob tag only has sense in a transition whose source is a branch state, and if this one has more than one transition going out of it.
 - If there are several transitions with no PAprob tag associated and going out of a branch state warn the user that the value that will be used is 1//number of transitions.
 - Warn the user if there's a transition with PAprob tag and is not going out of a branch state.
 - PAinitialState is a tag that must be in an state.
 - Every state diagram must have only one initial state.
 - If there are calls to events, these must exist and have the same name as the one by they are called. This is, again, a restriction imposed by the way ArgoUML works.
 - Warning, check if every state with a do-Activity has a PArespTime tag, if not warn the user.
 - PAsize is a tag that must be in an event.

- *Deployment Diagrams:*
 - The classes listed in the GRMmapping tag must exist in the model.
 - The classes only can reside in physical nodes.
 - Between physical nodes must be communication nodes, which must be stereotyped and have PAspeed tags.
- *Class Diagrams:*
 - PApopulation is a tag that must be in a class.

Once the model has been validated it's passed to the model processor that will translate it into Petri Nets, analyze it and return the results.

4.4 Model Processor

Once we have a configured XMI file containing a model, that won't raise user errors during the process time we must convert the model to Petri Nets, analyze it and show the user the information he wanted. The Model Processor [6, Cap9] is in charge of make those things. As it's specified in the Profile [6, Cap9], the model processor is divided in three parts:

- The Model Convertor, that will convert the model into whatever model to be analyzed, in our case Petri Nets.
- The Model Analyzer, that will analyze de model.
- Result Convertor, that will return the results in a way that they can be showed using, in our case, the ArgoUML tool.

To communicate each part of the Model Processor a ProcessingModel is used, like Model was used with parts above (Model Editor, Model Configurer and Model Processor). This ProcessingModel class will store the configured XMI model, and the query that the user has made, all the intermediate steps and transformation and last the results of the query.

It must be said that the model processor needs one value for each annotation (except from the GRMcode ones). So the decision about which values of the multiple ones must have been made by de user previously.

It must be said too, that if the xmi model was processed before it's not processed again, the results are directly copied to the Model.

4.4.1 Queries the module can answer

Before commenting the different parts of the ModelProcessor the things that the module can analyze must be commented.

The queries that the module allows the user to choose are the following (summarized):

- *Stay Time*: This query will return the time, in seconds that are spent in an state from an state machine of a class.
- *Time in State*: This query will return the percentage of time that is spent in an state selected by the user.
- *Transmission Speed*: When two classes, nodes or class and node are selected, we can query what speed are they connected at, measured in KBps.
- *Message Delay*: When a transition with an event is selected, we can calculate, in seconds, what is the delay since the event is called until it arrives at the class waiting for that event.
- *Just Translate*: This query only translates the UML model into the model used for the translation, in our case, Stochastic Petri Nets in GSPN format.

4.4.2 Model Convertor

The Model Convertor main purpose is to convert the configured XMI model that receives into something that can be analyzed, like Petri Nets. Our Model Convertor will return the Petri Nets in GSPN format.

This part receives the parsed XMI file, and traverses it, finding the diagrams. When a diagram is found then the corresponding diagram translator is called. The abstract generation class is PetriNetBuilder, that implements the methods to write the Petri Nets into the files that GreatSPN uses. That class also defines the method that any diagram translator must have. All those methods are TXXX where T means translate and XXX is the element, for example, TClass or TNode or TState.

So, the conversion process starts finding the diagrams and calling the translators, SMtoGSPN, ClassInfoGetter or DeploymentInfoGetter are the main classes used, because the lack of Sequence diagrams in ArgoUML.

When all this process ends the result is a ComponentPetriNets results, storing not only the Petri Nets for each element in the model but logistic information about the classes, states, events and so on, and information about the connection between the components of the model.

As it was said before, not always a unified petri net is needed, for instance, to calculate the Message Delay of an event we only need the information about the connection of the classes involved and the size of the event. So, there are times that only the ComponentPetriNets results are stored in the ProcessingModel.

If we want to calculate any other, more complex thing, such as Time in an State, we must merge this ComponentPetriNets into one Petri Net of the whole system. This only petri net is stored then in the ProcessingModel.

To merge an handle all the PetriNets the edu.cps.perforance.modelprocessor.model.greatspn.GreatSPNHelper class is used. This class uses the GreatSPN tool or tools that that one provides, such as algebra, to manage the nets, merge them or analyze them.

4.4.3 Model Analyzer

The Model Analyzer receives the ProcessingModel containing the Component-PetriNets and, if needed, one PetriNet in GreatSPN format of the whole system. Then, depending on the query that the user made, the model is analyzed.

In every case this model analyzer uses the `edu.cps.perfomrance.modelprocessor.model.greatspn.GreatSPNHelper` that implements all the methods that can be used to calculate all the queries, except from the Message Delay query that doesn't need to deal with the Petri Nets to achieve its information.

So, the performance of this part is simple, finds what query is to be done, gets the information needed from the ProcessingModel class and calls the corresponding method to calculate that, that will be implemented in the GreatSPNHelper or as a private method in the Concrete Model Analyzer.

The results of the analysis are stored once again in the ProcessingModel class, that will be passed to the concrete ResultConverter by the Model Processor, that is the one in charge of uniting the different parts and making them work as a complete Model Processor.

4.4.4 Results Converter

The result converter is the part that will translate the results back to the interface we're using to show those results to the user.

In our case, the results are quite simple, Integers or a Vector, so the results are copied without translation from the ProcessingModel to the Model class, and then they will be showed in ArgoUML.

5 Execution Flow

In this section we'll present how the whole module works. We will explain, since the moment the user chooses one query to made, what methods, parts of the module are invoked until the results are showed back to the user.

5.1 PAModuleAction

When the user clicks in the ArgoUML menu selecting then what query he wants to make the class PAModuleAction is called, to be more precise, the *model-editor. argouml. connection. actions. PAModuleAction: actionPerformed()* method in that class.

The actionPerformed method deals with all the parts of the module and controls their works.

We present now the work flow in the actionPerformed:

- *Step 1:* Checks the query to be make, if it's TimeInState, StayTime, TransmissionSpeed, MessageDelay or JustTranslateModel. Then checks if the user has selected what must be selected for the query, for example, if

the user has selected `TimeInState`, an state must have been selected. If the user has selected `MessageDelay` a transition with an event must have been selected, and so on. If all is correct we pass to the next point, if no the execution stops informing the user to select what he must. This checks are made via private methods in this `PAModuleAction` class.

- *Step 2:* Call to the `ModelEditor` to generate the `Model`. This called will return the class `Model` with the maybe parameterized XMI model.
- *Step 3:* Call to the `ModelConfigurer`, to configure the model if it's needed and to generate the parsed XMI and the `annotationTable` always. If there's no problems configuring the model the we pass to the next step. Problems that may occur are, wrong config file, parameters with no value, wrong annotations and so on. If there are problems the execution stops here.
- *Step 4:* Call the `UML Validator`. If the `Model` is correct the pass to the next step. If the model has errors inform the user and exit. If the model has warnings we warn the user that the results may not be correct and ask the user if he wants to continue. If so we pass to the next step, if not we stop here.
- *Step 5:* Check if the model has multiple value annotations, and if so make the user choose which ones he wants to use. Once he has chosen create a new `annotationTable` with the values chosen by the user and store it in the `Model` class.
- *Step 6:* Call to `ModelProcessor` to process the `Model`. This will return the results stored in the `Model` class.
- *Step 7:* Call to the private method `showResults`, that will show the results with different messages depending on the query that was made. If there were multiple value annotations we ask the user to use other values in the annotations, if he wants to finalize now the execution is stop, if not we go again to Step 5.

In the following sections we will present what is done in the steps above.

5.2 Step1: User selects query

User has clicked in one of the available options in the tool-performanceanalysis menu in ArgoUML. The option chosen is passed in the `ActionEvent` parameter that is passed to the `actionPerformed` method in the `PAModuleAction`. We check the name of the event to know what query the user selected.

Once we know the query the user made we need to get the parameters needed for the query. For example, if the query is time in state we need to get the state, if the query is transmission speed we need to get two classes, two nodes or a node and a class. We get this via private methods implemented in the `PAModuleAction`, such as `selectState` or `selectEvent`. This methods uses

the ArgoUML source code to traverse the diagrams and find out which elements are selected. If those methods don't return the element, but null or something like that, the user didn't select the elements needed, so it's informed with an ErrorMessage in ArgoUML and the execution stops here.

If everything was right we store the query that the user has chosen, the elements selected by the user and go to next step.

5.3 Step2: Model Editor

In this step we tell the Model Editor to generate the class Model that will be used in the whole performance analysis process. This class needs to have now the XMI file of the UML model. The *edu.cps.performance.modeleditor.argouml.xmi.ConcreteArgoUMLXMIModelEditor* is the class that will create the model class and generate the XMI model using its method generateModel(). In this method the Model class is created and then the concrete XMIGenerator is called to create the XMI file. In our case the concrete XMIGenerator, the class that extends the *edu.cps.performance.model.xmi.XMIGenerator* class is the *edu.cps.performance.modeleditor.argouml.xmi.ArgoUMLXMIGenerator*. We use the method getXMIFile() of that class that returns a java File class containing the XMI file. The getXMIFile() method checks if the model has been transform into XMI before. If so the XMI previously calculated is return, if not a protected method, saveProjectToXMI() is called, and is the one that really writes the model into the XMI file. As it was said before this model performs the same actions as the export to XMI plug-in that comes with the ArgoUML source code. Any change in the ArgoUML code may make the export to XMI plug-in change, so it will be useful check the plug-in source code every time we update our ArgoUML version.

Once the file is generated the ConcreteArgoUMLXMIModelEditor stores it in the *edu.cps.performance.model.Model* class and returns that Model class.

The Model class contains only the XMI file, but we need to store all the information that we got in step1, so a private method insertSelectedParamsInTheModel() is called in the actionPerformed() method in the PAModuleAction class.

5.4 Step3: Model Configurer

Now we have the class Model containing the XMI file, that may be parameterized, the query the user chose and the elements selected by the user needed to analyze that query.

This step is in charge of configuring the XMI file. The configuration is done by the *edu.cps.performance.modelconfigurer.ConcreteXMIModelConfigurer*. This class checks if the Model has been configured before. If so we ask the user to change the configuration file that was used, or just return the previous configured Model. If the Model wasn't configured before we proceed with the configuration process.

The steps to configure the model are the following:

1. Parsing the XMI File. That will return a DOM tree and a dictionary. This is done by the *edu.cps.performance.model.xmi.XMIParser* that uses the xerces package to parse XML.
2. Configuring the Model. We check first that the parsing is correct and now we call the *edu.cps.performance.model.xmi.XMIConfigurer*. This makes all the configuration as it's explained in section XXX. The AnnotationTable with all the annotations of the UML model is returned. To write the configured XMI file, the XMIConfigurer uses the *edu.cps.performance.modeditor.argouml.xmi.ArgoUMLXMIGenerator* that was used before.

If errors configuring the model arise, the user is informed and the analysis process ends.

Now the XMI model is configured, parsed and all its annotations stored in a table. We saved the DOM tree, dictionary, configured XMI file and the AnnotationTable in the Model class and go to next step.

5.5 Step4: UML Validator

Now the configured model with its annotations is passed to the UML Validator, that will check if the UML model is correct. The checks that this part makes are listed in section XXX.

The validation process is directed by the *edu.cps.performance.validation.ConcreteArgoUMLValidator* class. This class traverses the DOM tree, checking each element that is found. For example, if we find an state, the checkState() private method is called, and there we check its annotations, its name and everything that needs to be checked.

Every time an error or a warning is found we insert it in the *edu.cps.performance.validation.ValidationErrors* class that will be returned when the validation process ends.

Once the validation process has finished we check if there are errors, if so we show them and stop the performance analysis process. If there weren't errors, but there were warnings we show them and ask the user if he wants to continue. If so we go to next step.

5.6 Step5: Selection of Multiple Values

We have now a configured XMI model, parsed, with its annotations and checked so that the process will not raise user errors.

Now, if the model has multiple values to one stereotype-tag the user must choose which one use. This is made in the actionPerformed method. We get the AnnotationTable of the whole model, and get the multiple values in it and we create an AnnotationTable with the stereotypes and tags that only had one value. We store the multiple values in an

edu.cps.performance.util.SelectionValueDialog which will be presented to the user. The user will choose then the values to use. This values will be stored in the AnnotationTable created with the single values, and then stored in the model. Then the model is passed to the Model Processor.

To avoid processing the same model with the same annotations many times we use a CacheResults class in *edu.cps.performance.util*. There we store the AnnotationTables and the results that have been calculated, and if the user wants to analyze the model again with a set of values, we search first in the CacheResults class, if the result was calculated we return it, if not we go to the next step.

5.7 Step6: Model Processor

We have now the Model with an AnnotationTable with only single values (except from the classes resident in a node).

The process followed here is similar to the one in the *actionPerformed()* method. The *processModel()* method in the *edu.cps.performance.modelprocessor.ConcretePNModelProcessor* will direct the work of the different parts of the process.

- *Step 1* Create a ProcessingModel class, that will be passed to the different parts in the process.
- *Step 2* Call to the *convert()* method in the concrete Model Convertor class.
- *Step 3* Call to the *analyze()* method in the concrete Model Analyzer class.
- *Step 4* Call to the *convert()* method in the concrete Result Convertor class.

Here we'll explain the work flow in those steps.

5.7.1 Creating ProcessingModel

Just like the Model class was created to be passed to the different parts of the Performance Analysis process, a ProcessingModel class is created here with the same purpose, be passed to the different parts in the process.

The *edu.cps.performance.modelprocessor.model.ProcessingModel* is created with the information stored in the Model class, that is passed to the Model Processor class. We store in the ProcessingModel the parsed XMI file and AnnotationTable, that will be passed to the next step in the process.

5.7.2 Convert ProcessingModel

That UML model, stored as XMI and parsed must be now converted to Stochastic Petri Nets, that we'll store in the GreatSPN file format.

This function is implemented in the *edu.cps.performance.modelprocessor.modelconvertor.greatspnconvertor.ConcreteGreatSPNModelConvertor* and is made by the *convert()* method.

First thing that's done here is check if the ProcessingModel has been converted before. If so we return the converted model and store it in the ProcessingModel. If not a new conversion is needed.

First step in the conversion is the getting the Model information and generating of the component petri nets. This information is made by the generateComponentPetriNets in the *edu.cps.performance.modelprocessor.modelconverter.greatspnconverter.ConcretePNTranslator* class, called in the convert() method of the ConcreteGreatSPNModelConverter. The result of that method is a *edu.cps.performance.modelprocessor.modelconverter.greatspnconverter.results.ComponentPetriNet* class, that, besides storing the Petri Nets of each State Machine, class etc in the model, stores the Model information such as communication info or logistic info (names, events and so on).

This conversion is made in a similar way to the validation previously explained. We traverse the DOM tree finding the different elements to be translated, and that will generate a Petri Nets or whose information is needed (Classes or State Machine diagrams or Deployment Diagrams). For each one of those a PetriNetBuilder is created. For example, for Classes a ClassInfoGetter, for Deployment, DeploymentInfoGetter or for State Machine diagrams an SMtoGSPN. These classes traverse again the DOM tree, and for each element they need the implement one method to get the information of the element and use it to translate if necessary. For example, if the SMtoGSPN finds an state then the TState() is called and it will get the name, annotations of the state and create one (or more) states in the Petri Net equivalent to the one that has been found in the State Machine Diagram. Another examples is, when the DeploymentInfoGetter finds a node then TNode() method is called and the info about if the node is a physical or communication one, or the classes resident or the speed is got. Then all that information is used to create the petri nets, the transmission costs diagrams, etc. The ClassesInfo class stores the info of the classes including its translated Petri Net. The ModelInfo class stores the ClassesInfo information and the transmission costs information of the whole model. And all the nets and the ModelInfo is stored in the ComponentsPetriNet result. The ComponentsPetriNet result is not one Petri Net of the whole model, but a set of Petri Nets of the components of the UML model. We store the ComponentPetriNet in the ProcessingModel class.

Once the ComponentsPetriNet is calculated we analyze what the query the user wanted to calculate is, and then, if the query needs one Petri Net of the whole model a call to private method in the convert() method is done. This private method will make special markings in the Petri Net, depending on the query done and calls the GreatSPNHelper to calculate the merged Petri Net of the whole UML model. Then we store it in the ProcessingModel. The one net is a GreatSPNPetriNet class.

5.7.3 Analyze ProcessingModel

Now we have the UML model translated into an Stochastic Petri Net. We have stored the model in a ComponentPetriNets result and if it has been needed in a

GreatSPNPetriNet class all inside the ProcessingModel. No the analyze method is called. As we have said we use the GreatSPN tool to analyze the Petri Nets.

The first thing that's done in the *edu.cps.performance.modelprocessor.modelanalyzer.greatspnanalyzer.ConcreteGreatSPNModelAnalyzer* is retrieving what the query to be answer was. Once we have the query we call a private method in the ConcreteGreatSPNModelAnalyzer that will store in a Results class the results. For example, if the query is analyze Time in State then the `_getResultTimeInState()` method in the ConcreteGreatSPNModelAnalyzer is called. This methods returns a TimeInStateResult, that is stored in the Results class and then saved in the ProcessingModel class.

Inside the private methods different analysis are made. For the state queries the GreatSPNHelper class is called to retrieve the time in state results. The calculations are made and the result is given. For calculating the Message Delay or TransmissionSpeed the information is in the ComponentPetriNet results, in the ModelInfo part. The transmission costs and size of events are stored there, so we got them, use them for calculations and store the result.

That's how this part works, so each query is calculated with the information of the analysis of the GreatSPNPetriNet or from the conversion process, where information of the model was stored.

Now we have saved the results in the same class, the *edu.cps.performance.model.results.Results* class, and this class is stored in the ProcessingModel.

5.7.4 Return Results of the ProcessingModel

This part of the process must translate back the results in a way that they can be showed in the ArgoUML tool. Due to the fact that we use only Floats, Integer or Vectors as results no translation is needed. If we need some day to save files or something like that maybe a translation of those results would be needed.

Now this part of the process Model only copies the Results in the ProcessingModel to the Model class. That's done in the `convert()` method in the *edu.cps.performance.modelprocessor.resultconverter.argouml.ConcreteArgoUMLResultsConverter*.

When we end this part the results are stored in the Model class, and then we return to the `actionPerformed()` method again.

5.8 Step7: Show Results

The last thing done in the `actionPerformed()` method is to show the results that the analysis has returned. This is done calling a private method in the PAModuleAction. This method is `showResults`. In this method we find our what query was to be answered, we get the results of the Model and create a MessageDialog to show the results.

As it was said, ff there were multiple values in the model we ask the user if he wants to try with other values. If so we find out in the ResultCache class if

the Model was previously analyzed with that values. If not we go again to step 6. If the Model has been analyzed then the Result is stored in the ResultsCache, we get it and show it as we have said.

If the user doesn't want to continue analyzing the model with new multiple values or the model hasn't got any then the performance analysis has ended.

6 Further Work

Here we'll present what things can be done in the future to improve the Performance Analysis module:

- Use another CASE tools instead of ArgoUML. That is a good idea because another tools like Rational Rose has implemented all the diagrams that we may need, more queries can be implemented, an the modelling in Rose is far better than the one that we can do now in ArgoUML.
- Translate the model to new Petri Nets formats, instead of using the non standard GreatSPN format to store petri nets. We can make another module in the ModelConvertor to translate to PNML format.
- Improve the translation to Petri nets, including multiple values during the translation. In my opinion, this new feature is not recommended to be done by a new person to the module, because it's very hard, and it needs a total control over the conversion process.
- Check the translation probes made to the module. This involves making the translation in paper and then checking that this translation is equal to the one made by the module.

References

- [1] Rational Software Corporation., 2001. <http://www.rational.com>.
- [2] The ArgoUML project. <http://argouml.tigris.org>.
- [3] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language*. Addison Wesley, 1999.
- [4] The GreatSPN tool. <http://www.di.unito.it/~greatspn>.
- [5] The Java Compiler Compiler. <https://javacc.dev.java.net>.
- [6] Object Management Group. *UML Profile for Schedulability, Performance and Time Specification*, March 2002. <http://www.omg.org>.
- [7] Petri Nets Markup Language. <http://www.informatik.hu-berlin.de/top/pnml/about.html>.