
Plasma Lab Documentation

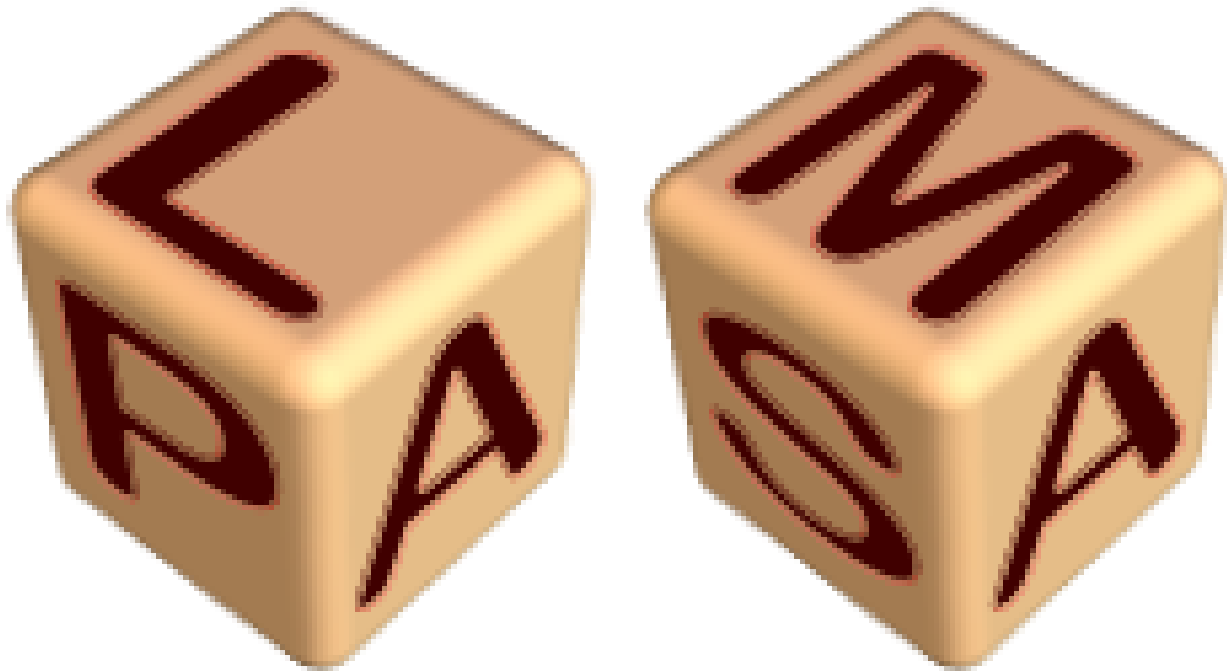
Release 1.4.4

Louis-Marie Traonouez, Kevin Corre, Matthieu Simonin

Apr 18, 2018

CONTENTS

PLASMA LAB



PLASMA Lab is a compact, efficient and flexible platform for statistical model checking of stochastic models. PLASMA Lab demonstrates the following advances:

- Use your own simulator and checker via our plugin system.
- Build your software around Plasma Lab using our API.
- Prism (Reactive Modules Language-RML) and Biological languages supported.
- Matlab, LLVM, SytemC plugins.
- Distributed architecture. Whether you plan to use several computers on a local area network or a grid, you can run PLASMA Lab in an easy way.
- Fast algorithms.
- Efficient data structure, low memory consumption.
- Developed with Java for compatibility.

PLASMA Lab, is being integrated into the DALi, ACANTO and DANSE project platforms.

1.1 How to cite PLASMA Lab

- **Statistical Model Checking of Simulink Models with Plasma Lab.** Axel Legay, Louis-Marie Traonouez, FTSCS 2015: 259-264
- **PLASMA-lab: A Flexible, Distributable Statistical Model Checking Library.** Benoît Boyer, Kevin Corre, Axel Legay, Sean Sedwards, QEST 2013: 160-164
- **A Platform for High Performance Statistical Model Checking – PLASMA.** Cyrille Jégourel, Axel Legay, Sean Sedwards, TACAS 2012: 498-503

1.2 People

- **LEGAY Axel** – Team leader – axel.legay[-at-]inria.fr
- **TRAONOUÉZ Louis-Marie** – Developer – louis-marie.traonouez[-at-]inria.fr
- **QUILBEUF Jean** – Developer – jean.quilbeuf[-at-]inria.fr

1.2.1 Former members

- **BOYER Benoit** – Developer
- **CORRE Kevin** – Developer
- **JEGOUREL Cyrille** – Contributor
- **NGO Van-Chan** – Developer
- **SEDWARDS Sean** – Developer – sean.sedwards[-at-]inria.fr
- **SIMONIN Matthieu** – Developer – matthieu.simonin[-at-]inria.fr

1.3 Download

PLASMA Lab is developed with Java 7. Previous versions may present issues and are not supported. The main distribution bundles for the latest version 1.4.4 of PLASMA Lab can be downloaded on the download page of the website:

<https://project.inria.fr/plasma-lab/download/>

These bundles include PLASMA Lab GUI and PLASMA Lab CLI, the Graphical and Command Line Interfaces, all the libraries needed to run PLASMA Lab's SMC Algorithms, including the libraries to run distributed experiments, and a selection of plugins. Older and experimental versions, specific plugins can also be downloaded from the website.

Download and extract one of these files and run them in a terminal you should observe this structure :

```
.
+-- configs          # contains the configuration files
+-- demos            # contains some examples
+-- libs             # all the libs needed by plasma to run
+-- plugins          # contains the plugins (you can add your own here)
+-- plasmacli.bat    # (windows) starts the cli
```

(continues on next page)

(continued from previous page)

```
+-- plasmacli.sh      # (*nix) starts the cli
+-- plasmagui.bat     # (windows) starts the gui
+-- plasmagui.sh      # (*nix) starts the gui (plasmalab or service)
+-- plasmaservice.bat # (windows) starts the service gui
+-- README.md
```


GRAPHICAL INTERFACE

This chapter describes how to use the Graphical User Interface of Plasma Lab. We also presents PLASMA Service, the distributed client of PLASMA Lab. Finally we describes the configuration options of PLASMA Lab.

2.1 Open and Edit a project

PLASMA Lab Graphical User Interface (GUI) is composed of several elements:

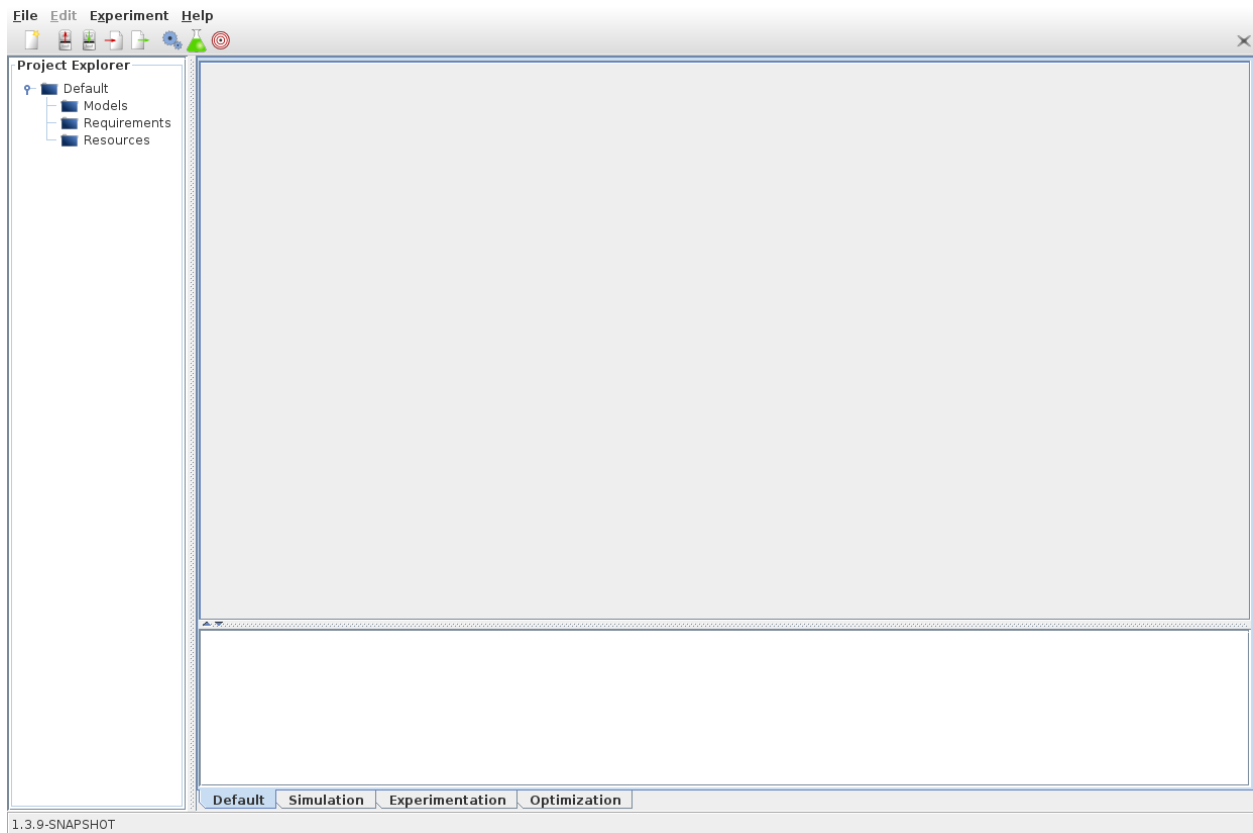
- A **main tabbed panel**, that displays projects and experimentation/simulation panels.
- A **project explorer**, that shows the tree structure of opened projects.

Start the GUI :

```
# \*nix - from the command line
./plasmagui.sh launch

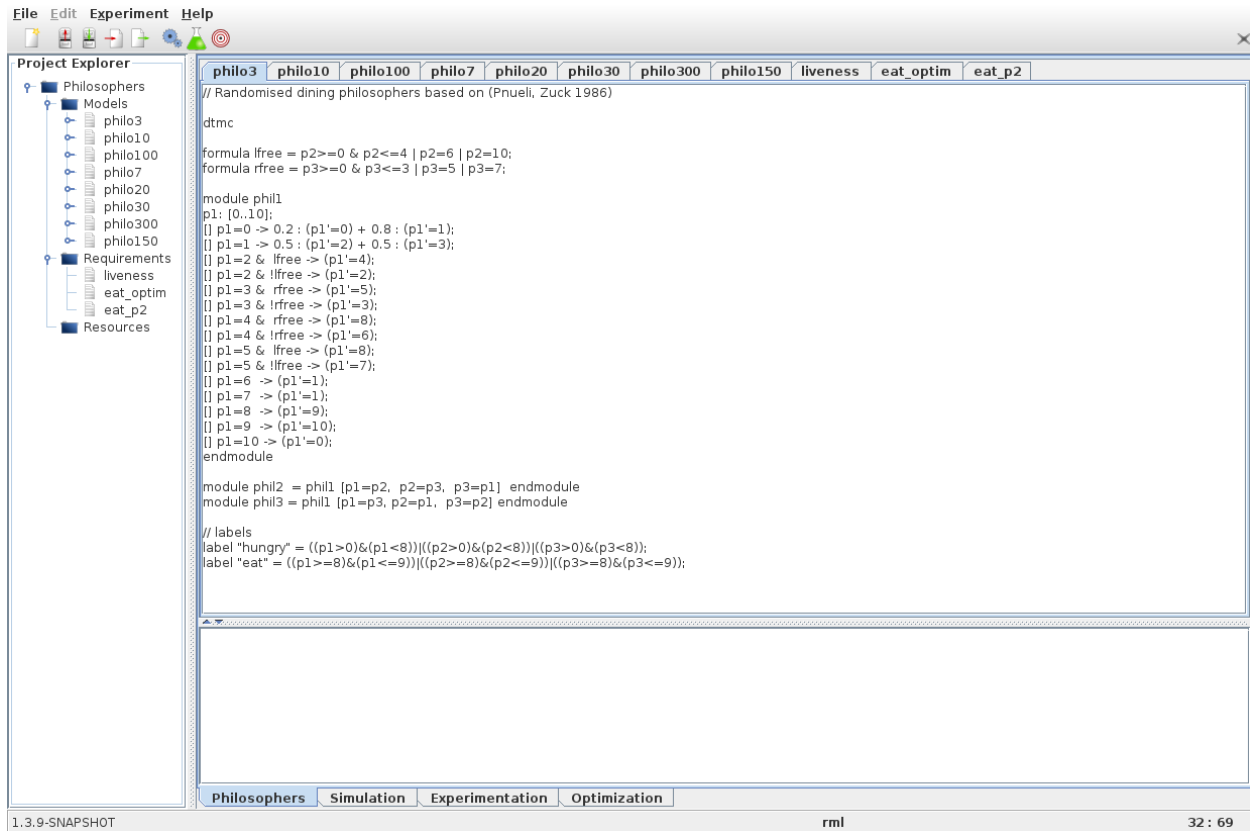
# windows - double click on
plasmagui.bat
```

This is the windows you will see after launch:



We will have a later look at the experimentation and simulation panels. Let's focus on the project panel. In this panel we can edit two types of data. **Models** that describe our system and **Requirements** that we want to verify.

Download this [sample project](#) and open it in PLASMA Lab. This can be done using the File menu. You can also find more examples [here](#).



You can now edit your model and properties. You could also create new properties and models or import them using the File menu. **Be sure to select the right type when creating a new data or importing one.** If you want to rename a file you can do so by right-clicking on the item you want to rename in the project explorer. At the bottom of the edit panel you can find **error messages** that concern the properties and models you are editing as well as an indication showing **the type of the selected data**.

The next page shows you how to test your model in the *simulation panel*.

2.1.1 PLASMA Projects

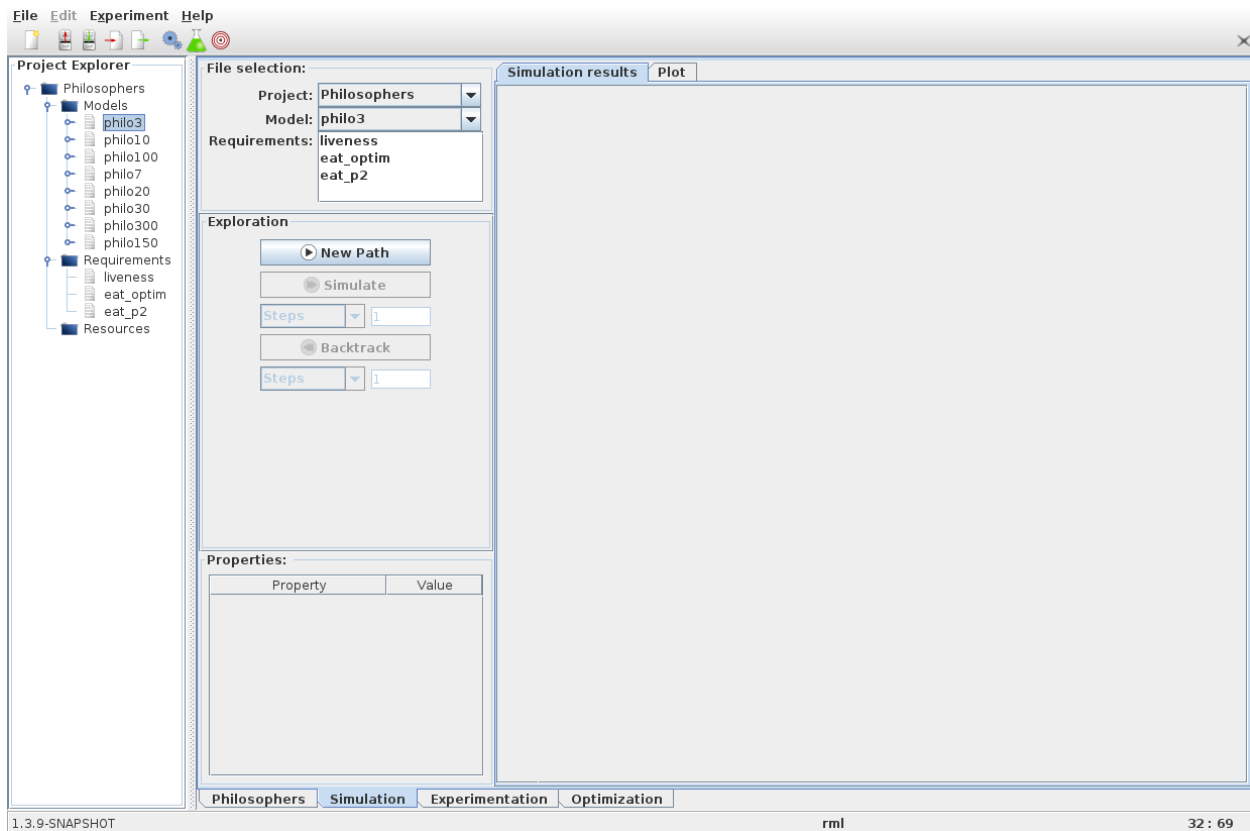
Project files (extension .plasma) are XML files that allows to build a collection of models, requirements and possibly other text resources. Legacy project files are self-contained: all the models or requirements are included in the XML file as a text field.

From version 1.4.2 project files can also be links to existing files. In that case saving the project in PLASMA Lab GUI will save the content of the models and requirements on the files given in the project file. When creating a new project, if a content is added with the **New** menu the content will be saved in the XML file. If a content is imported with the **Import** menu from an existing file, the content will always be saved in that file, and only a path to the file will be saved in the XML file. Note that a project can contain at the same time content from the XML file and content from a file.

2.2 Simulation mode

The **Simulation** panel allows to generate one trace step-by-step, and to look at the values of the model's variables and the properties along this trace. It is made of four different areas:

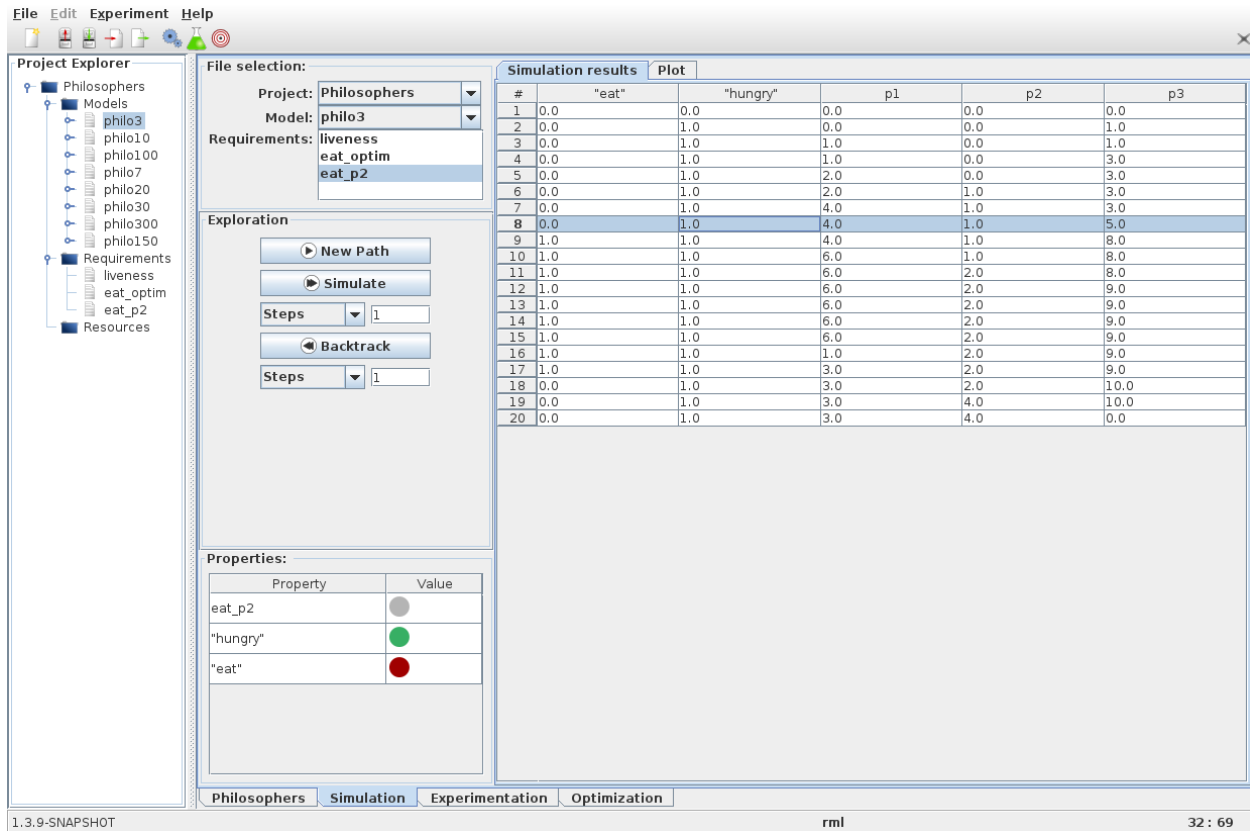
- A *File selection* panel
- An *Exploration control* panel
- A *Properties* panel
- A *Simulation results* panel



In order to simulate our model we select the project and the model from the *File selection* panel. We can also select a property to check during the simulation, but this is not mandatory.

To start a new trace we use the **New path** button in the simulation control panel. The initial state of our model appears in the simulation result panel. We can then **Simulate** the model or **Backtrack** using the corresponding buttons.

When the simulation progresses or when we select a specific state, values are updated in the property panel.



Alternatively to the *Simulation results* panel, we can use the *Plot* panel. This panel allow us to draw **2D or 3D plots**. To draw a 2D plot showing the evolution of p1 against time select the blank value in the X combo box (the empty identifier represents the discrete time) and p1 in the Y combo box.

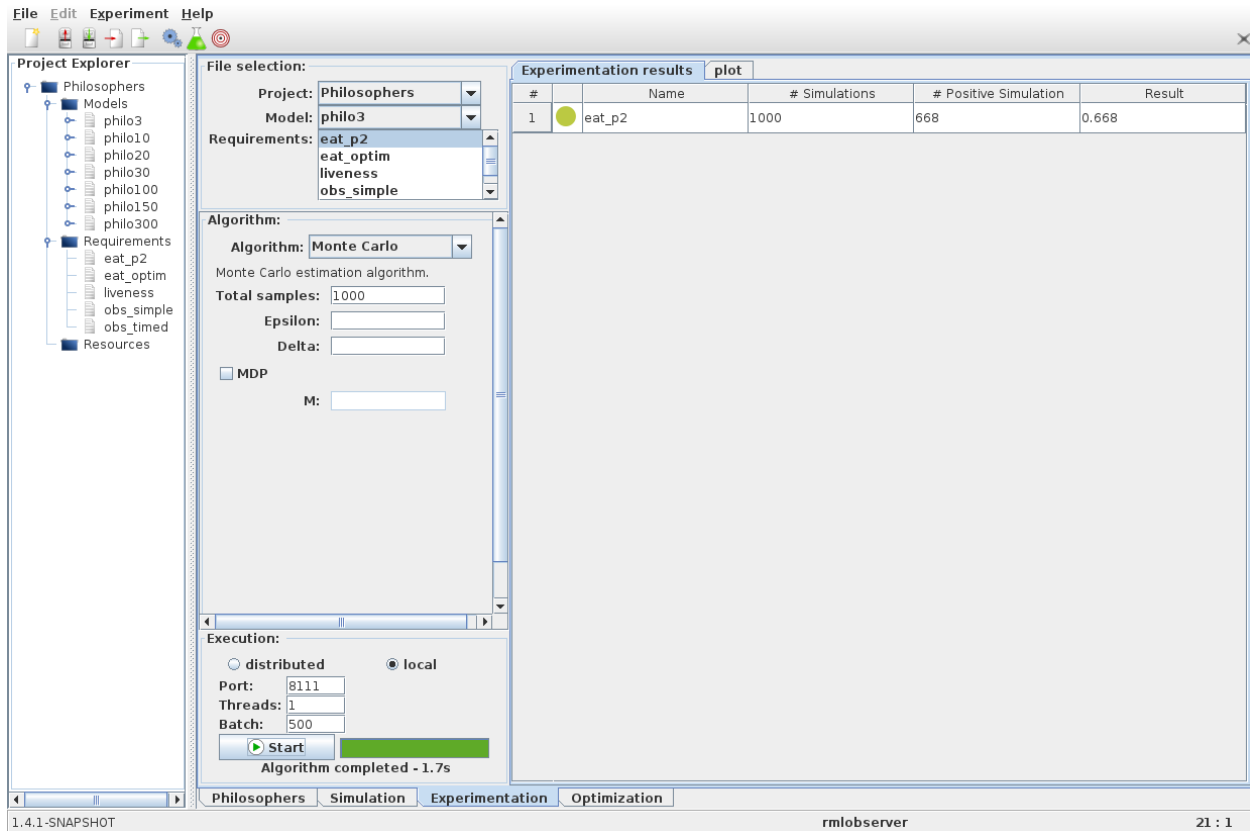
Once we have tested our model and properties, we can go to the *experimentation panel* to use PLASMA Lab SMC algorithms. This is the next step of our tutorial.

2.3 Experimentation mode

The **Experimentation** panel is made of 4 different areas:

- A *File selection* panel
- An *Algorithm* panel
- An *Execution* panel
- An *Experimentation results* panel

As previously, we select the project and the model from the *File selection* panel as well as requirements to check (at least one). We choose an SMC algorithm in the *Algorithm* panel and configure its parameters. For this tutorial lets choose the Monte Carlo algorithm with a *Total samples* parameter of 1000 simulations. Finally we click on the *Start* button at the bottom of the interface. Once the 1000 simulations have been computed, results show up in the result panel.



We can also select the *Plot* panel to have a better understanding of our results by drawing plots.

The next step of the tutorial presents the optimisation panel.

2.4 Optimization mode

The **Optimization** panel extends the experimentation mode by allowing a user to define a set of initial states. When launching an experiment each initial states will be checked as a separate experiment.

Defining an optimization variable depends on the model or property language. It is possible to define optimization variables in a property, which will modify the associated model.

Compared to the *Experimentation* panel, the *Optimization* panel has an additional panel to modify optimization variables before running an experiment.

#	Name	p1	# Simulations	# Positive Simulati...	Result
1	eat_optim F<=#5...	0.0	1000	120	0.12
2	eat_optim F<=#1...	0.0	1000	849	0.849
3	eat_optim F<=#5...	1.0	1000	283	0.283
4	eat_optim F<=#1...	1.0	1000	903	0.903
5	eat_optim F<=#5...	2.0	1000	594	0.594
6	eat_optim F<=#1...	2.0	1000	946	0.946
7	eat_optim F<=#5...	3.0	1000	597	0.597
8	eat_optim F<=#1...	3.0	1000	952	0.952
9	eat_optim F<=#5...	4.0	1000	860	0.86
10	eat_optim F<=#1...	4.0	1000	978	0.978
11	eat_optim F<=#5...	5.0	1000	887	0.887
12	eat_optim F<=#1...	5.0	1000	986	0.986

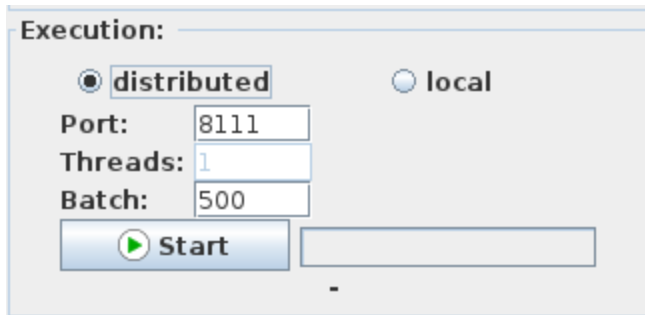
2.5 Distributed experiments

PLASMA Lab implements parallel SMC algorithms that allows to massively distribute the simulations. In distributed mode, PLASMA Lab GUI acts as a server that controls the SMC experiment while a PLASMA LAB client is launched on a distant computer to perform simulations. The server requests simulations from the clients and then wait for the results. Finally it aggregates the different results and it either shows the final result in the gUI or it requests more simulations if needed.

2.5.1 Setup a PLASMA Lab distributed experiment

When setting our experiment in the previous tutorial we didn't used the *Execution* panel that is set to local by default. This panel allows to configure 3 different executions mode:

- A local single thread mode that is used by default when the selector is on local and the number of threads is equal to 1.
- A local multi-threaded mode when the selector is on local and the number of threads is at least 2.
- A distributed mode that waits for clients to connect to the experiment.



In the last two modes, the number of **Batch** defines the maximum number of simulations that will be performed in a row by a client.

In distributed mode we can configure the port on which PLASMA Lab is listening.

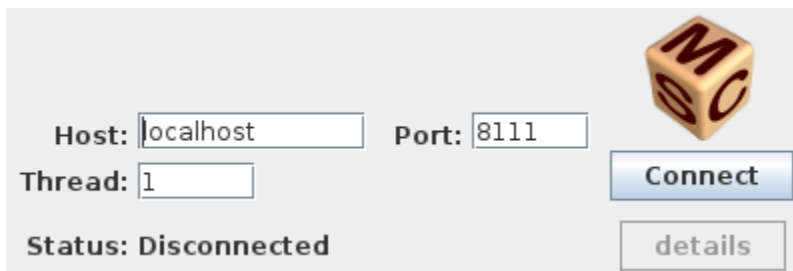
We now select the distributed mode and then click on the *Start* button. This will initialize the server and start the algorithm scheduler.

2.5.2 Setup PLASMA Service

In order to run a distributed experiment we must connect clients to the server in order to perform the simulations. PLASMA Service is a small GUI that starts a client:

```
# \*nix - from the command line
./plasmagui.sh service

# windows - double click on
plasmaservice.bat
```



PLASMA Service lets you define the host address and the port on which PLASMA Lab is listening. You can also set the number of threads that you want to run on this instance of PLASMA Service.

Once PLASMA Service is setup we click on the connect button. PLASMA Service status now switch to running. Once our experiment is completed, PLASMA Service disconnect itself.

Alternatively you can run PLASMA Service in a command line interface. This might be useful if you intend to run it on a distant machine.

2.6 Configuration

PLASMA Lab GUI can be launch with the following options:

```
Usage: launch [options]
Options:
  --conf
      Path to Plasma Lab configuration file.
  -d, --debug
      Debug level
      Default: 0
  --help
      Print help.
      Default: false
  --log
      Use a logback xml file
```

PLASMA Lab use a text configuration file to load plugins and set parameters. If no configuration file was provided, a default one is loaded from the jar resources.

Here is the list of configuration:

- plugin ABSOLUTE_PATH_JAR

This option will tell PLASMA Lab to look for plugins in the jar file located at ABSOLUTE_PATH_JAR and load them.

- plugin_dir ABSOLUTE_PATH_DIR

This option will tell PLASMA Lab to look for every plugins located in the ABSOLUTE_PATH_DIR directory and load them.

- working_dir RELATIVE_PATH_DIR

This option will set RELATIVE_PATH_DIR as the working directory. The working directory is the default directory when loading or saving files.

- log4j RELATIVE_PATH_XML

This option tell PLASMA Lab to load the log4j configuration file located at RELATIVE_PATH_XML. In the case where this option was not set, a default log4j configuration file is loaded from the jar resources.

COMMAND LINE INTERFACE

This chapter describes how to use the Command Line Interface (CLI) of Plasma Lab.

Starts the CLI :

```
# \*nix - from the command line
./plasmacli.sh

# windows - from the command line
plasmacli.bat
```

Several sub-commands are available and will be explained in the following (shown with the `plasmacli.sh` script, but working equivalently with `plasmacli.bat`). They are specified on the command line using `./plasmacli.sh subcommand`.

You can always invoke the help for a subcommand using :

```
./plasmacli.sh subcommand --help
```

3.1 Info

`./plasmacli.sh info [options]` provides information option about PLASMA Lab's plugins and algorithms.

- `./plasmacli.sh info` displays the list of plugins (algorithms, models, requirements) that is currently loaded by PLASMA Lab

(with the plugin list defined in the configuration file). For each plugin it gives the ID of the plugin, that should be used when running the CLI to identify the plugin, and a short description.

- `./plasmacli.sh info -a [algorithm id]` displays information about an algorithm, in particular the list of parameters.

3.2 Launch

`./plasmacli.sh launch [options]` is used to start a new experimentation. You'll have to pass all the parameters required to perform the experimentation (requirement with option `-r`, model with option `-m`, algorithm with option `-a` and its parameters with options `-A`).

Models and requirements files must always be written with their type using the following syntax: `file:type`.

Example :

```
./plasmacli launch -m demos/raw/philos3.rml:rml -r demos/raw/liveness:bltl -a montecarlo -
↪A"Total samples"=10000 --progress
```

In this case you should observe the following output

```
[...]
[some traces]
[...]
[=====] 100%
+-----+-----+-----+-----+
| Name      | # Simulations | # Positive Simulation | Result |
+-----+-----+-----+-----+
| liveness  | 10000         | 10000                 | 1.0    |
+-----+-----+-----+-----+
```

3.3 Simu

`./plasmacli.sh simu [options]` is used to start an interactive simulation. A necessary option is the model to simulate.

- Pressing Enter will step the simulation by 1 step (initial default value)
- Entering a number `n` will step the simulation by `n` steps. Then `n` will be the new default value for the step number.
- Entering `r` restart the simulation at the initial step.
- Entering a negative number allows to backtrack the simulation.
- Entering `q` will quit the simulation.

3.4 Service

`./plasmacli.sh service [options]` is used to start a new service listening on the default port.

LANGUAGES

PLASMA Lab uses two types of data, models and requirements. On these pages you will find details on these languages. You could also take a look at our collection of [examples](#).

4.1 Models

In its current version, PLASMA Lab is bundled with three model languages:

- Reactive Module Language (RML) of the PRISM tool, a popular probabilistic model checker tool supporting a wide range of models. More documentation on this language can be found on [PRISM website](#).
- *Adaptive Reactive Module Language* is an extension of RML for adaptive systems.
- *Biological Language*.

Additionally, three plugins allow to interface PLASMA Lab with external tools:

- *MATLAB/Simulink*
- *SystemC/MAG*
- *LLVM*

As we are using a modular architecture, you can also develop your own model language and integrate it into Plasma Lab.

4.1.1 Adaptive Reactive Module Language

We have extended RML with structures that model adaptive systems. We detail below the modifications and new syntax added to RML.

Modules

Modules can now be parameterized using the following construction:

```
module name ( parameter1, parameter2, parameter3, ... )  
  // declarations  
  ...  
endmodule
```

The parameters syntax is `parameter ::= type name`, and they can be of the following types:

- `const int`, `const double`, `const bool` for constant parameters respectively integers, double and Boolean.

- `int`, `double`, `bool` for variable parameters respectively integers, double and Boolean. Variable parameters are also local variables. Note that therefore they must be renamed by the renaming operation.

Parameterized modules define templates that can be instantiated using the following command:

```
module new_name = name ( expression1, expression2, expression3, ...) endmodule
```

This creates a new module *new_name* that instantiates a parameterized module of type *name* such that each parameter is instantiated by the value of the corresponding expression. In case a parameter is variable, this instantiation corresponds to fixing the initial value of the variable. The new module is a copy of the parameterized module such that all the variables (and parameters) in the new module are renamed, their new name being their old name prefixed by *new_name*.. Note that parameterized modules can also be renamed using the classic renaming command from RML. This operation creates a new parameterized module.

Systems

The system construction from RML is extended in Adaptive RML. Several systems can now be designed. Each of them defines a different configuration of the adaptive system. A system can also be parameterized.

```
system name ( parameter1, parameter2, parameter3, ...)
// declarations
...
endsystem
```

The syntax of the parameters is the same as for modules. However, only constant parameters can be used. The declarations in a system consist in a set of module instantiations. Each of them uses the syntax previously presented. Contrary to RML in which the system always consists in all the modules of the model, in Adaptive RML each system may contain a different set of modules. Note that non parameterized modules are also added to a system using the same syntax, only in that case the expressions list is empty. Additionally, a declaration in a system can be a formula or a label, i.e. an expressions over the variables of the system, using the same declaration command from RML.

Adaptive Systems

Finally, the adaptive system is declared as a set of adaptive command, i.e. meta-transition from one system to another. The syntax is the following:

```
adaptive
  init at start_system;
  // adaptive_commands
  ...
endsystem
```

start_system is a system instantiation: it consists in a system name and a list of expressions to instantiate the parameters of the system. Adaptive commands are written in the following syntax:

```
{ system ( | guard )? } -> ( probability : {new_system} )+;
```

system specifies the current system in which the command is enabled. Additionally a guard can be specified, that is an expression over the variables of the system. Then comes a list of actions, each of them composed by a probability and the definition of the new system. The probabilities are expressions over the variables of the current system. The sum of the values of the probabilities must be lower than one. *new_system* is a system instantiation that defines the next configuration reached by the adaptive system.

4.1.2 Biological language

Here is the BNF grammar of our biological modeling language:

```

BIO ::= ["constant" ConstantList]
      "species" SpeciesList
      ReactionList

ConstantList ::= ConstDecl [", " ConstantList]

ConstDecl ::= ident "=" Number

SpeciesList ::= SpecDecl [", " SpeciesList]

SpecDecl ::= ident [ "=" integer]

ReactionList ::= Reaction [ReactionList]

Reaction ::= Reactants [RateConst] "->" Products

Reactants ::= "*"
            | Species [ "+" Species [ "+" Species ] ]

Species ::= ident

Products ::= "*"
           | Species { "+" Species }

RateConst ::= Number
            | ident

Number ::= integer
         | float

```

4.1.3 MATLAB/Simulink

In order to use SMC methods on wider and more complex models, we worked on interfacing PLASMA Lab with MATLAB/Simulink.

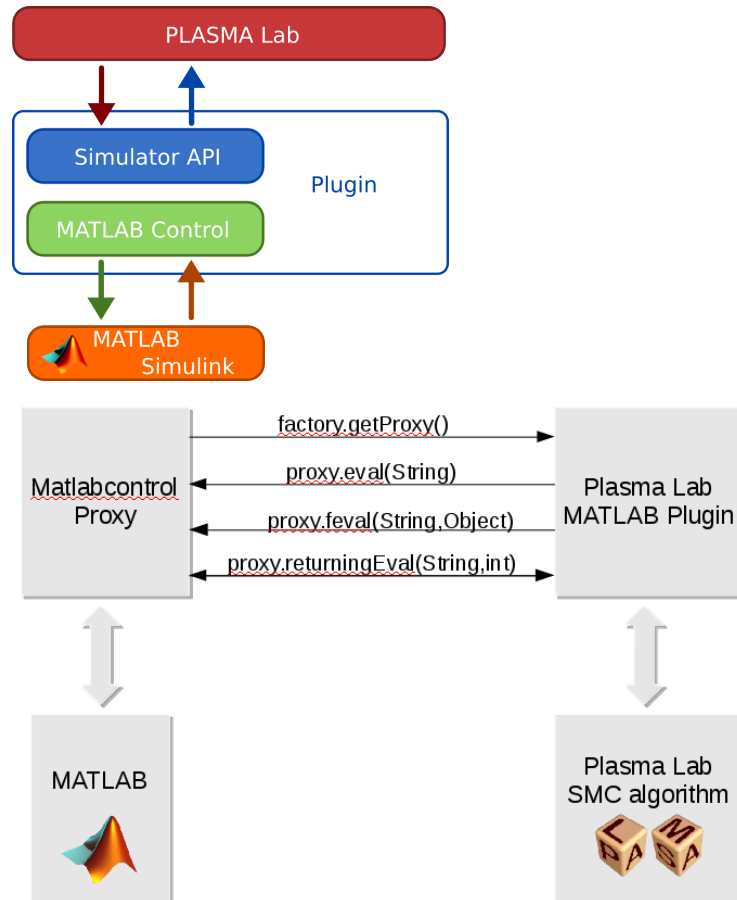
We first describes how we implemented the interface between PLASMA Lab and Simulink, and give a short explanation on how to use the Simulink plugin in PLASMA Lab. Then, in the second section, we detail the PLASMA2Simulink App than can be installed in MATLAB, how to install it and how to use it.

Simulink plugin for PLASMA Lab

The Simulink PLASMA Lab plugin is developed in the project *fr.inria.plasmalab.matlab* and it implements the Simulator API.

Simulink proxy interface

To connect Simulink to PLASMA Lab we use the [MATLAB Control library](#). This API allows to interact with MATLAB from Java. A proxy object, MATLABProxy can be used to send commands to MATLAB (like *eval*, *feval*), and to get and set variables.



This library allows us to execute the same functions we would have used in MATLAB to run simulations. We use to run a simulation with the Simulink simulator of MATLAB. This produces a trace with the signals that have been declared as output in Simulink as well as the global time. This trace is retrieved in Java and stored in an InterfaceState object.

The following code for instance is used to create the trace:

```
// SIMULATE @ fr.inria.plasmalab.matlab.MatLabSessionModel

proxy.eval(plasmaOutput+" = sim(idSML, "
            + "'StopTime', num2str("+simTimeLength+"), "
            + "'StartTime', num2str("+simLatestTime+"), "
            + simParameters +");");
```


To emulate the *state on demand* approach we took in PLASMA Lab design, we simulate the Simulink model for a parameterized amount of time and store the latest state. If more states are needed to decide the requirement, the simulation can be extended from the latest state. If not, a new run is executed from the initial state. The length of each simulation run is parameterized by the *Stop time* parameter of the Simulink model.

Although we are using MATLAB Control to communicate with MATLAB, that is a general library to control MATLAB, our PLASMA Lab plugin can only execute Simulink models due to its implementation.

How to use it

To use our Simulink plugin in PLASMA, create a new MATLAB Interface model in PLASMA Lab. Once created, the plugin will launch MATLAB. It is not currently possible to connect to an existing MATLAB session.

We specify the absolute path to the Simulink model in the PLASMA Lab edition panel. Opening the model can take some time as it requires MATLAB to load the Simulink library.

A configuration step must be done on the Simulink model, but it is straightforward. The PLASMA Lab plugin will only trace signals set as output. To log a signal apply the following steps:

1. Select a the signal link in the Simulink model and right click on properties.
2. Give a name to the signal.
3. Check the box *Log signal data* under the tab *Logging and accessibility*

Also you should edit the model properties:

1. Open the *Model configuration parameters* panel (with a right click on the model or Ctrl+E)
2. Go to the *Data Import/Export* section
3. Under *Time, State, Output* set the *Format* to *Structure with time*
4. Under *Signals* set *Signal logging format* to *DataSet*

Each time more states are needed, Simulink will move the simulation forward from the latest state for a parameterized amount of time. We use the *Stop time* parameter of the Simulink model for this purpose.

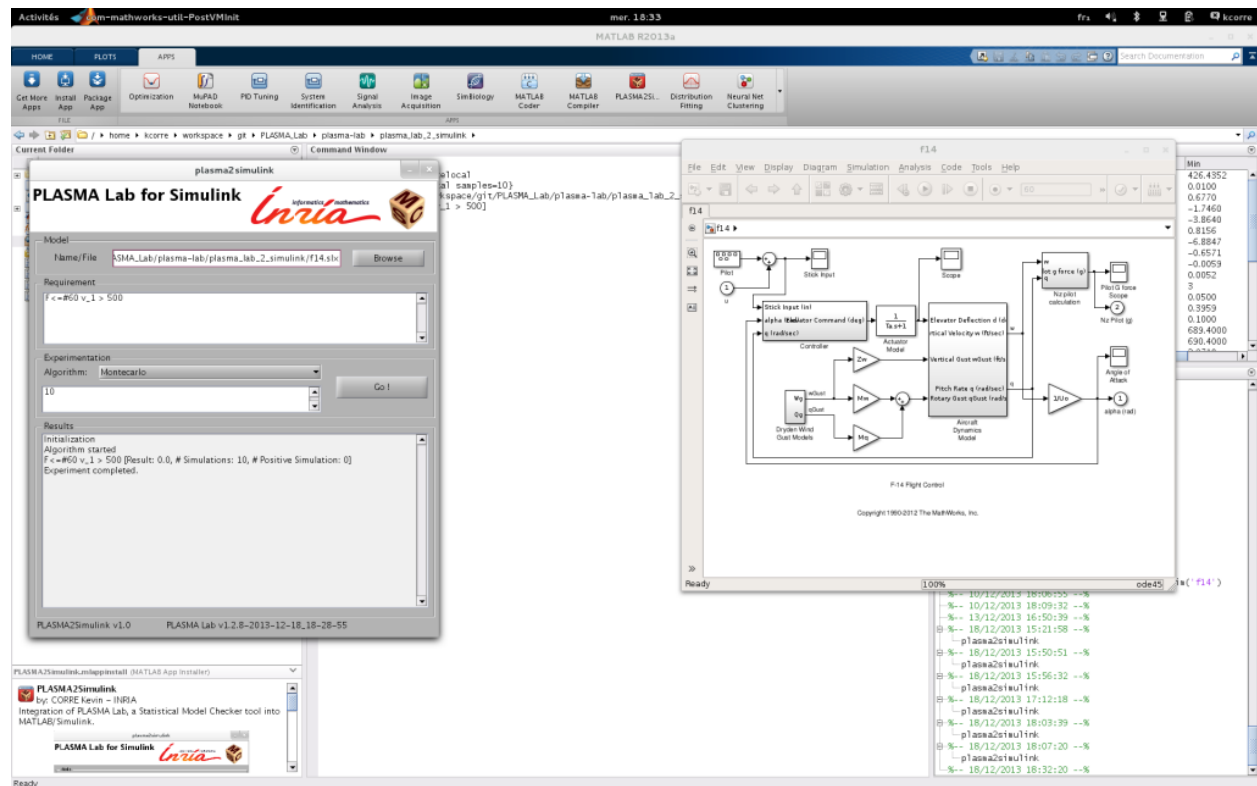
Once our model is configured, we can test it in the Simulation panel and then launch an experiment.

This Simulink plugin is not yet compatible with the distributed mode.

PLASMALab2Simulink

The PLASMALab2Simulink App is a MATLAB App aimed at using PLASMA Lab directly in MATLAB. The App provides a small user interface written with the MATLAB language. It also contains PLASMA Lab libraries with the SMC algorithms, the Simulink plugin and the BLTL requirement plugin.

The *fr.inria.plasmalab.matlab_ui* project contains a code layer communicating with the MATLAB code. Sources for the MATLAB part of the code can be found in the *plasmalab2simulink* directory. The build procedure for the App is documented in the Continuous Integration section of this manual.



How to use it

To use the PLASMA Lab2Simulink App, download the .mlappinstall file on the download page and open it with MATLAB. This will install it on your MATLAB installation and a PLASMA Lab2Simulink icon will be added to the Apps tool bar.

To configure a Simulink model, follow the same procedure described in the *Simulink plugin, How to use it* section. The App has less functionality than the regular PLASMA Lab GUI. To launch a new experiment specify the path to the Simulink model and enter a property using the BLTL syntax.

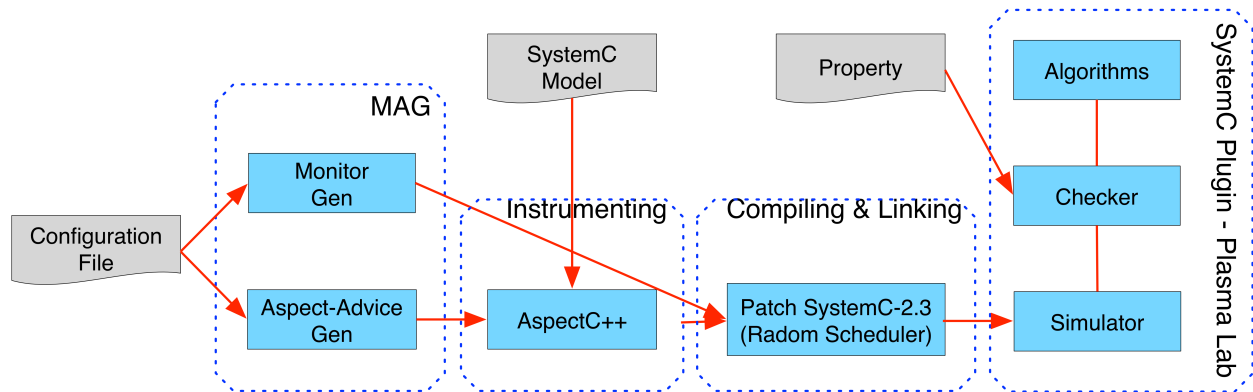
Algorithms parameter are set in a single text field using white-space as separator. Parameters should be given in the same order as in the PLASMA Lab GUI experimental panel.

- **Montecarlo** : Total samples
- **Chernoff** : Epsilon Delta
- **Sequential** : Alpha Beta Delta Proba

4.1.4 SystemC

SystemC is a high-level modelling language for specifying concurrent processes. It is implemented as a set of C++ classes that allow to perform event-driven simulation. Probabilistic behaviors can also be added.

We have implemented a SystemC plugin for PLASMA Lab that is able to load a SystemC executable model and use it to generate simulations. PLASMA Lab and the SystemC plugin are embedded in the toolchain of the [Probabilistic SystemC Verifier \(PSCV\)](#) tool:



This manual explains how to use PLASMA Lab’s SystemC Plugin and the MAG tool in order to perform statistical model checking on SystemC models. Running the verification framework consists of two steps:

1. Generating an executable model corresponding to the verification of properties using the MAG tool
2. Using the SystemC plugin plugin to verify the properties.

Download

The following tools are required to use the SystemC plugin:

- Monitor and Aspect-advice Generator: [MAG 1.0](#)
- Aspect-oriented Programming with C/C++: [AspectC++ 1.2](#)
- Patched version of SystemC in [CHIMP 1.0](#)

Installation

To compile and install MAG, AspectC++ and the patched version of SystemC, refer to the [MAG user guide](#).

Generating Executable Model with MAG

In order to run the SystemC plugin to verify properties, users first need to generate an executable model from the original SystemC model. It consists of three steps:

1. Use MAG to generate the monitors and the aspect-advice file.
2. Run AspectC++ with the generated aspect-advice file on the generated monitors and original model code to instrument the original model code.
3. Compile (e.g., with g++) the instrumented SystemC model code to produce the corresponding executable model

Users run MAG to generate the monitors and aspect-advices according to the properties verified as follows:

1. Write the configuration file according to the properties verified (see the [MAG user guide](#) for more details)
2. Change to the directory of the MAG tool and run `mag -conf "configuration file"`.
3. MAG generates three files: header and source files of the generated monitor, and the aspect advice file `aspect_definitions.ah` in the location defined in the configuration file.

Users run AspectC++ with the aspect-advice file on the original model source code and the generated monitors to generate the instrumented SystemC model.

1. Change the directory of AspectC++ 1.2

2. Run the following command to generate `puma.config` file in the working directory: `ag++ -gen_config`
3. Copy the header and source files of the generated monitors and the aspect-advice file `aspect_definitions.ah` into the source directory of the original model source code
4. For each header or source file of the SUV, run the following command to generate the instrumented version:

```
ac++ -c source_path/input_file -o target_path/output_file -p source_path -I source_path -I systemc_path/include -config aspectc_path/puma.config
```

where `source_path` is the path to the source directory of the original model source code, `target_path` is the path to the directory where the AspectC++ puts the instrumented version, `systemc_path` is the path to the patched version of `systemc-2.3.0` (included in CHIMP), and `aspectc_path` is the path to the AspectC++ 1.2.

Compiling the instrumented model source code is done as follows:

1. In the main header file, include the generated monitor header file, for example `#include monitor_multi_lift.h`
2. In the main source file, add the following line just before the call to `sc_start()`:

```
mon_observer* obs = local_observer::createInstance(1, other_parameters);
```

The parameters depend on the generated monitor, for example, in the included example of multi-lift system in the MAG tool package, it is:

```
mon_observer* obs = local_observer::createInstance(1, &liftsystem);
```

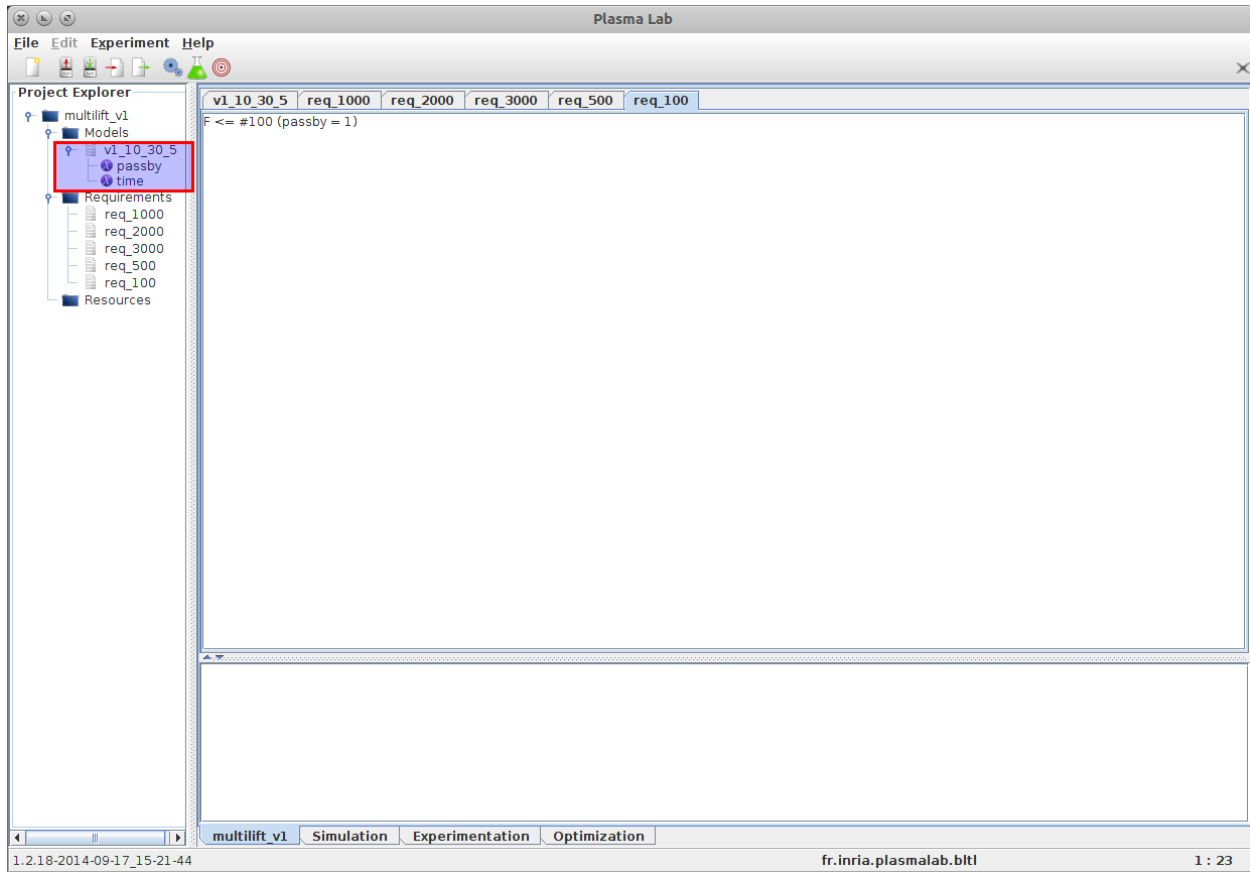
3. Compile the instrumented model source code with `g++` compiler and link it with the patched `SystemC`

We also provide shell scripts in the example directory of MAG tool that automatically generate the instrumented model source code. User can modify them according to their configuration and requirements.

Running the SystemC Plugin within PLASMA Lab

After compiling the instrumented `SystemC` model, an executable model is produced. Users can then verify the desired properties with the `SystemC` plugin of Plasma as follows:

1. Launch PLASMA Lab with the `SystemC` plugin loaded.
2. Create a new model of the type *SystemC Specification*. The content of the model is the path to the executable model
3. Write the properties to verify, based on the observed variables that are available in the executable model, as shown in the following figure (inside the red area):



4. Check the properties in the *Experimentation* panel

4.1.5 LLVM

From version 1.4.3, PLASMA Lab includes a plugin that simulates LLVM code. This plugin is a wrapper of an external simulator [Lodin](#). This tool is necessary to use the plugin.

To use the PLASMA Lab plugin we create a model of type LLVM and we write in the content of the model the two following lines:

- The first line must contain the path to the Lodin simulator (the `sim` binary) followed by the options to use.
- The second line must contain the path to the LLVM code to simulate (the `.ll` file compiled from C code).

Some Lodin options are mandatory to run with Plasma Lab:

- The `-X` option to use the alternative state output.
- The `-C` to run Lodin in a command based protocol.

Additionally the option `-S` is mandatory to run Lodin with the importance splitting algorithm, as it allows to store the states of the simulator in order to restart simulations.

4.2 Requirements

Requirements languages also are modular. Plasma Lab comes with five:

- *Bounded-LTL (BLTL).*
- *Adaptive BLTL (ALTL).*
- *Goal Contract Specification Language (GCSL).*
- *Nested BLTL.*
- *Observers*

4.2.1 Bounded Linear Temporal Logic

Our property logic is based on a Bounded Linear Temporal Logic (B-LTL). This logic enhance Linear Temporal Logic (LTL) operators with bounds expressed in step or time units. These bounds give the length of the run on which the nested formula must hold. Any decidable property on states or runs can be used in the formulas including nested B-LTL operators. Thus, the semantic of our B-LTL logic is the semantic of LTL logic restricted to a time interval.

For a quick introduction to LTL you could read the Wikipedia article on [Linear Temporal Logic](#).

B-LTL Grammar

The grammar accepted by our BLTL plugin is the following:

```
Property ::= ( ('declare' Variable ';' Variable)*)?
            ('optimize' Variable ';' Variable)*)?
            'end')? Formula

Formula ::= 'F <=' Bound Formula
          | 'G <=' Bound Formula
          | Formula 'U <=' Bound Formula
          | Formula 'W <=' Bound Formula
          | 'X' ('<=' Bound)? Formula
          | Formula '&' Formula
          | Formula '|' Formula
          | Formula '=>' Formula
          | '!' Formula
          | '(' Formula ')'
          | 'true'
          | 'false'
          | Numerical Operator Numerical

Bound ::= ('#')? Numerical

Variable ::= ident ':' (Interval | number)

Interval ::= '[' number ';' number ';' number ']'

Numerical ::= ident | number

Operator ::= '<' | '<' | '!=' | '=' | '>=' | '>' | '+' | '-' | '*' | '/'
```

The difference with classical Linear Temporal Logic is that temporal operators (F, G, X, U and W) are bounded by a temporal bound. This bound may either be a number of steps (using the syntax `<= # Numerical`) or a real-time bound (using the syntax `<= Numerical`). If the model is untimed, the two syntax are equivalent.

Optimisation

It is possible to declare a new variable in a BLTL property and assign a range of values to it. This generates a set of BLTL formulas that can be checked simultaneously by some SMC algorithms. The syntax is `declare variable:=[min;max;inc] end`, where *variable* is the name of the variable, *min* is the minimum value assigned to the variable, *max* is the maximum value assigned to the variable, *inc* is the increment between each instantiation.

For instance:

```
declare K:=[5;10;5] end
```

```
F<=#(K) "eat"
```

generates a property whose temporal bound is successively 5 and 10.

It is also possible to optimize existing variables in the model and assign similarly a range of initial values to it. When checking such a property several initial states will be successively checked by the SMC algorithm. The syntax is `optimize variable:=[min;max;inc] end`, where *variable* is the name of the model's variable, and *min*, *max*, *inc* are as previously.

For instance:

```
optimize p1:=[0;5;1] end
```

```
F<=#5 "eat"
```

will check the property for each values of the variable p1 from 0 to 5 with an increment of 1.

4.2.2 Adaptive BLTL (A-BLTL)

We have extended the grammar of BLTL with two adaptive operators that observe adaptive transitions in stochastic adaptive systems (SAS). The grammar of the new logic is the following:

```
Property ::= ( ('declare' Variable (';' Variable)*)?
              ('optimize' Variable (';' Variable)*)?
              'end')? Formula

Formula ::= BLTL_Formula
          | BLTL_Formula ';' [' Trigger ',' Trigger ']' ==>' Formula
          | BLTL_Formula ';' [' Trigger ',' Trigger ']' <=' Bound '==>' Formula
          | BLTL_Formula '==>' [' Trigger ',' Trigger ']' ;' Formula
          | BLTL_Formula_Formula '==>' [' Trigger ',' Trigger ']' <=' Bound ';' Formula

Trigger ::= Trigger '=>' Trigger
          | Trigger '&' Trigger
          | '!' Trigger
          | '(' Trigger ')'
          | 'true'
          | 'false'
          | Numerical Operator Numerical
```

(continues on next page)

(continued from previous page)

```
Bound ::= ('#')? Numerical

Variable ::= ident ':= ' (Interval | number)

Interval ::= '[' number ';' number ';' number ']'

Numerical ::= ident | ident '"' | number

Operator ::= '<' | '<' | '!=' | '=' | '>=' | '>' | '+' | '-' | '*' | '/'
```

As presented above, the two adaptive operators of A-BLTL can be either bounded or unbounded. Informally the semantics of these operators is the following:

- BLTL Property `'==> [' Trigger ',' Trigger '] ;' Property`, is satisfied by an execution if the first BLTL property is satisfied by the first state and there exists an adaptive transition that satisfies the triggers and the second property is satisfied by the state reached just after the adaptive transition.
- BLTL Property `' ; [' Trigger ',' Trigger '] ==>' Property`, is satisfied by an execution if when the first BLTL property is satisfied by the first state and there exists an adaptive transition that satisfies the triggers, then the second property is satisfied by the state reached just after the adaptive transition. Therefore in this semantics it is not necessary to observe an adaptive transition in order to satisfy the property.

The triggers are Boolean expressions over the variables of the model. The first trigger is evaluated on the state just before an adaptive transition, while the second is evaluated on the state reached just after the adaptive transition. To specify expressions, A-BLTL include an additional prime operator on identifiers that test if a variable exists and returns false in the contrary.

4.2.3 Goal Contracts Specification Language

As part of the DANSE project we are implementing a Goal Contracts Specification Language (GCSL). GCSL is a high-level language designed to use à la SPEEDS patterns.

The idea is to use a requirement language more understandable than raw logic. GCSL requirements would then be translated into a low-level logic that can be used by Plasma Lab.

Reactive Module Implementation

We made a prototype of the GCSL. As GCSL requirements are dependent on a model language, we implemented them to be used in conjunction with the Reactive Module language.

There are 9 patterns that we list here. On each of these examples (p, p_1, \dots, p_n) denotes a property, while **bold words** are keywords. Time intervals are denoted by $[a - b]$, where a and b are a number and a time unit. For now we only supports step as a time unit.

GCSL Grammar

```

GCSL ::=
| OCL-coll '->' 'forAll' '(' variable '|' pattern ')'
| OCL-coll '->' 'exists' '(' variable '|' pattern ')'
| OCL-prop
| pattern

pattern ::=
| 'whenever' '[' prop ']' 'occurs' '[' prop ']' 'holds' 'during' 'following' interval
| 'whenever' '[' prop ']' 'occurs' '[' prop ']' 'implies' '[' prop ']' 'during'
↳ 'following' interval
| 'whenever' '[' prop ']' 'occurs' '[' prop ']' 'does' 'not' 'occur' 'during' 'following'
↳ 'interval'
| 'whenever' '[' prop ']' 'occurs' '[' prop ']' 'occurs' ' within' interval
| '[' prop ']' 'during' interval 'raises' '[' prop ']'
| '[' prop ']' 'occurs' interval 'times' 'during' interval 'raises' '[' prop ']'
| '[' prop ']' 'occurs' 'at' 'most' '[' integer ']' 'times' 'during' interval
| '[' prop ']' 'during' interval 'implies' '[' prop ']' 'during' interval 'then' '['
↳ prop ']' 'during' interval

prop ::=
| OCL-prop
| expr ( '<' | '<=' | '=' | '>' | '>=' | '>' ) expr

expr ::=
| expr binop expr
| '(' expr ')'
| OCL-expr
| fun '(' expr ')'

fun ::= 'mean' | 'sum' | 'prod' | 'at'
binop ::= '+' | '-' | '*' | '/'

interval ::= lp time ('-' time)? rp
lp ::= '[' | '('
rp ::= ']' | ')'

time ::= int unit | '+oo'

unit ::= 'ms' | 's' | 'min' | 'hour' | 'day' | 'step' | ...

```

On these patterns we reused the OCL predicates **forAll** and **exists**. Given an instance of a module, these predicates will generate a list of all modules of the same “type” and will instantiate the pattern for this list. For instance, if we use the philosophers model in RML, we could define the following requirement in GCSL:

```

sys.itsphil1 -> forAll(phil | whenever [phil.p1 = 0] occurs [phil.p1 = 10] occurs within
↳ [5 step - 25 step])

```

The philosophers model in RML defines a module `phil1` and others modules `phil2` to `philn` by renaming variables of `phil1`. Thus, `sys.itsphil1 -> forAll(phil | p)` defines `phil` as a module of the same “type” as `phil1`.

Adaptive Reactive Module Implementation

The Adaptive RML is a RML extension to design cyber physical systems that may reconfigure themselves when the environment settings change. This capability is called as the dynamical adaptivity of the system and it can not be expressed easily using standard modelling languages. This extension is supported by the RML simulator in Plasma Lab.

When some changes of settings occurred, the properties that the system must guaranty may also change to be consistent with the new system configuration. It could be necessary to express some properties that where the properties before after the change are expressed in GCSL. We propose two adaptive patterns for that. They are based on the behavioral specifications (expressed in GCSL) before and after the change that is described using the a simple state property over the system to characterize the kind of change considered by the adaptive patterns. The grammar is given as

```
ADAPTIVE ::=  
| 'if' assumption 'holds' 'and' 'for' 'all' 'rule' 'that' 'satisfies' trigger 'then' '⌞'  
  ⇨ promise 'holds'  
| 'if' assumption 'holds' 'then' 'there' 'exists' 'a' 'rule' 'satisfying' trigger 'and' '⌞'  
  ⇨ promise 'holds'  
  
assumption ::= '[' GCSL ']'  
trigger ::= '[' prop ']'  
promise ::= '[' GCSL ']' | ADAPTIVE
```

The trigger is a simple state property, the assumption is written in GCSL and the promise can be a GCSL property or defined using an adaptive pattern too. When promise is used to nest several adaptive properties, the global property specifies properties over sequences of dynamical adaptive event in the system.

4.2.4 Nested B-LTL

We added a new **nested probability** operator. This operator allows you to check, within a property, if a sub-property's probability is superior or equal to a given probability. To do this we extended the B-LTL grammar with a new nested transition:

```
Property ::= 'Pr >=' number '[' Property ']'
```

ALGORITHMS

This section presents PLASMA SMC algorithms. The following algorithms are currently implemented. The *ID* is used to reference the algorithm in particular when using the command line. The *distributed* algorithms implements a separate server and client to distribute simulations.

Name	ID	Distributed
<i>Monte Carlo</i>	montecarlo	Yes
<i>Sequential</i>	sequential	Yes
<i>Comparison</i>	comparison	No
<i>Smart sampling</i>	smartsampling	Yes
<i>Seq. smart sampling</i>	seqsmartsampling	No
<i>Importance splitting</i>	splitting	Yes
<i>Cross entropy</i>	crossentropy	Yes
CUSUM	cusum	No

5.1 Monte Carlo methods

Monte Carlo methods allows to estimate the probability to satisfy a requirement. They can be used with confidence bounds like the Chernoff bound.

Monte Carlo methods can also be used to compute the average value of a reward property. In PLASMA Lab requirements may return float value that correspond to the result of some computation along the trace, like a cost or a reward. However, note that if these values or not bounded no confidence can be associated to the result estimated with Monte Carlo.

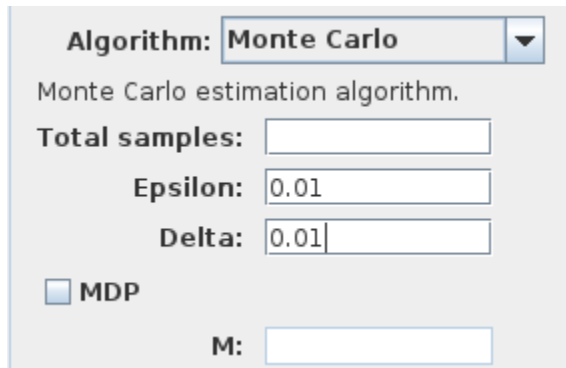
To run the Monte Carlo algorithm, either set the number of simulations:

- **Total samples** the number of simulations to run.

The screenshot shows a configuration window for the Monte Carlo algorithm. At the top, there is a dropdown menu labeled 'Algorithm:' with 'Monte Carlo' selected. Below this, the text 'Monte Carlo estimation algorithm.' is displayed. There are three input fields: 'Total samples:' with the value '10000', 'Epsilon:', and 'Delta:'. Below these fields is a checkbox labeled 'MDP' which is currently unchecked. At the bottom, there is an input field labeled 'M:'.

Otherwise the number of simulations can be determined using the Chernoff-Hoeffding bound based on two parameters:

- **Epsilon** the error margin.
- **Delta** the confidence bound.



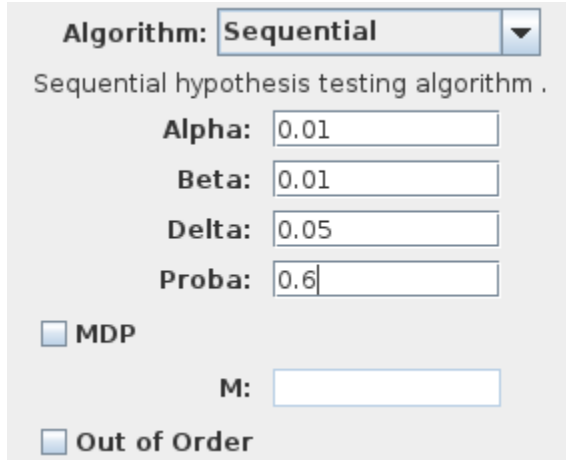
The screenshot shows a configuration window for the 'Monte Carlo' algorithm. At the top, 'Algorithm:' is followed by a dropdown menu set to 'Monte Carlo'. Below this is the text 'Monte Carlo estimation algorithm.'. There are four input fields: 'Total samples:' (empty), 'Epsilon:' (0.01), 'Delta:' (0.01), and 'M:' (empty). A checkbox labeled 'MDP' is checked.

MDP and **M** are used for *Markov Decision Processes*.

5.2 Hypothesis testing

5.2.1 Sequential Probability Ratio Test (SPRT)

The Sequential Probability Ratio Test (SPRT) is used check if the probability to satisfy a property is above or below to a given threshold. It needs four parameters :

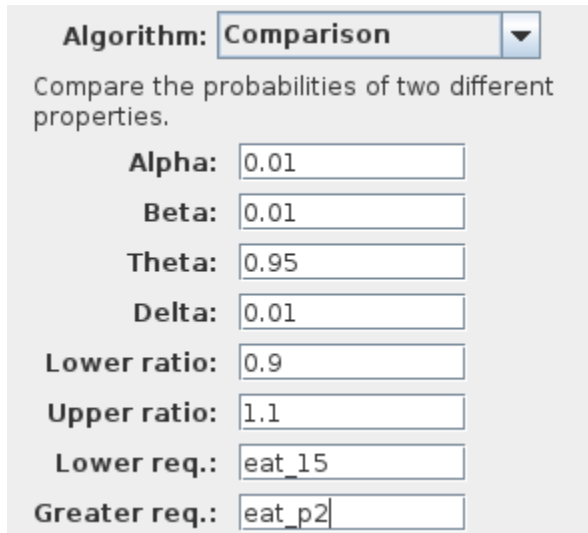


The screenshot shows a configuration window for the 'Sequential' algorithm. At the top, 'Algorithm:' is followed by a dropdown menu set to 'Sequential'. Below this is the text 'Sequential hypothesis testing algorithm.'. There are four input fields: 'Alpha:' (0.01), 'Beta:' (0.01), 'Delta:' (0.05), and 'Proba:' (0.6). A checkbox labeled 'MDP' is checked. Below the 'M:' field (empty), there is a checkbox labeled 'Out of Order' which is also checked.

- **Alpha** the false positives probability.
- **Beta** the false negatives probability.
- **Delta** the indifference region around proba.
- **Proba** the probability.
- **MDP** and **M** are used for *Markov Decision Processes*

5.2.2 Comparison algorithm

The comparison algorithm is used to compare the probabilities of two properties (whether one is greater than the other). Therefore, contrary to all other algorithms, it requires to select two requirements from the requirements list. The algorithm is based on an hypothesis test for the odds ratio of the two probabilities. It checks whether this ratio is lower or greater than one (with an indifference region). The algorithm also performs an additional hypothesis test to determine if the two probabilities are equal. Then it is configured with the following parameters:



Algorithm: **Comparison** ▼

Compare the probabilities of two different properties.

Alpha:

Beta:

Theta:

Delta:

Lower ratio:

Upper ratio:

Lower req.:

Greater req.:

- **Alpha** the false positives probability.
- **Beta** the false negatives probability.
- **Theta** the minimum probability needed to assess that the two probabilities are equal.
- **Delta** the indifference region around theta.
- **Lower ratio** the lower value for the odds ratio
- **Greater ratio** the greater value for the odds ratio
- **Lower req.** the name of the lower requirement
- **Greater req.** the name of the greater requirement

5.3 Probability estimation for MDPs

Markov Decision Processes (MDP) interleave nondeterministic actions and probabilistic transitions. Such models comprises probabilistic subsystems whose transitions depend on non the states of the other subsystems, while the order in which concurrently enabled transitions execute is nondeterministic. This order may radically affect the probability to satisfy a given property. It is therefore useful to evaluate the upper and lower bounds of these probabilities.

To calculate the expected probability of a sequence of states, it is necessary to define how the nondeterminism in the MDP will be resolved. We use the term scheduler to define such a strategy. Given a MDP with a set of action **A**, a set of states **S** and the set of sequences of states **T**, a history-dependent scheduler is a function from **T** to **A**. A memoryless scheduler is a function from **S** to **A**. We are interested in finding optimal schedulers that maximize or minimize the probability to satisfy BLTL properties.

5.3.1 Nondeterminism in RML

Nondeterminism naturally occurs when several transitions are enabled in the same state. If the model is declared using the keyword `dtmc`, PLASMA Lab will not consider the model as nondeterministic, but will apply a uniformly distributed choice between the different possible transitions.

MDP are the default model in RML. It can also be specified with the keyword `mdp` at the beginning of the model. By default (or by using the keywords `mdp sm1`) the model considers only **memoryless schedulers**. To consider **history dependent schedulers** the model must be declared with the keywords `mdp shd`.

In PLASMA Lab the default behavior to simulate and therefore check MDP models is to randomly select a scheduler for each new simulation. This behavior happens in PLASMA Lab simulator and in the basic algorithms (*Monte Carlo*, *Chernoff*, *Sequential*). Additionally, PLASMA Lab proposes several specific algorithms to deal with nondeterminism, which are described below.

5.3.2 SMC algorithms with multiple schedulers

The first approach to check nondeterministic models, and to compute minimal and maximal probability, is to consider a fixed number of schedulers, and to check each schedulers, using the classical Chernoff-Hoeffding bound or the Wald's sequential probability ratio test to bound the errors of the analysis. However, when iterating these tests for several schedulers, the probability to encounter an error will increase above the required error bound. Therefore PLASMA Lab uses specific error bounds to check MDP with several schedulers.

These error bounds are enabled in the *Experimentation* tab with *Monte Carlo* and *Sequential* algorithms by enabling the **MDP** box and entering the number **M** of schedulers.

The image shows two panels of the PLASMA Lab configuration interface. The top panel is for the 'Monte Carlo' algorithm, which is described as 'Monte Carlo estimation algorithm.' It includes input fields for 'Total samples' (empty), 'Epsilon' (0.01), and 'Delta' (0.01). There is a checked checkbox for 'MDP' and an input field for 'M' with the value 100. The bottom panel is for the 'Sequential' algorithm, described as 'Sequential hypothesis testing algorithm.' It includes input fields for 'Alpha' (0.01), 'Beta' (0.01), 'Delta' (0.05), and 'Proba' (0.6). There is a checked checkbox for 'MDP', an input field for 'M' with the value 100, and an unchecked checkbox for 'Out of Order'.

Algorithm: Monte Carlo ▼
Monte Carlo estimation algorithm.

Total samples:
Epsilon:
Delta:

☒ MDP
M:

Algorithm: Sequential ▼
Sequential hypothesis testing algorithm .

Alpha:
Beta:
Delta:
Proba:

☒ MDP
M:
☐ Out of Order

5.3.3 Smart sampling algorithms

The previous fixed sampling strategy of the set of schedulers has the disadvantage that it allocates the same simulation budget to each schedulers, regardless of their merit. To improve the performances of the analysis and maximize the probability of seeing a good scheduler, PLASMA Lab implements smart sampling techniques in three stages:

1. An initial undirected sampling experiment to discover the nature of the problem.
2. A targeted sampling experiment to generate a set of schedulers with high probability of containing an optimal scheduler.
3. Iterative refinement of the set of schedulers, to identify the best scheduler with specified confidence.

Smart probability estimation algorithm

The algorithm is available in the experimentation tab under the name **Smart sampling**. It computes either the **Minimum** or the **Maximum** probability. Like the classical Chernoff algorithm it allows to select the confidence **Delta** and the precision **Epsilon**.

It requires additionally a simulation **Budget** that will be used at each step. This simulation budget must be at least greater than the classical Chernoff bound. **Max budget** is the budget used at the second step of the algorithm to initialize the set of schedulers. It can be greater than **Budget** to test a wider range of schedulers. Otherwise, if empty it is equal to **Budget**. **Reduction factor** is a positive integer (greater than 2) that specifies the amount of schedulers that are eliminated after each iteration (equal to 2 if empty).

The *reward algorithm* is enabled when the name of a reward is inserted in the **Reward** field. The **SPRT** parameter is only used in that case.

Algorithm: Smart sampling ▼

Smart sampling Monte Carlo estimation algorithm for non-deterministic models.

☒ **Minimum**
☐ **Maximum**

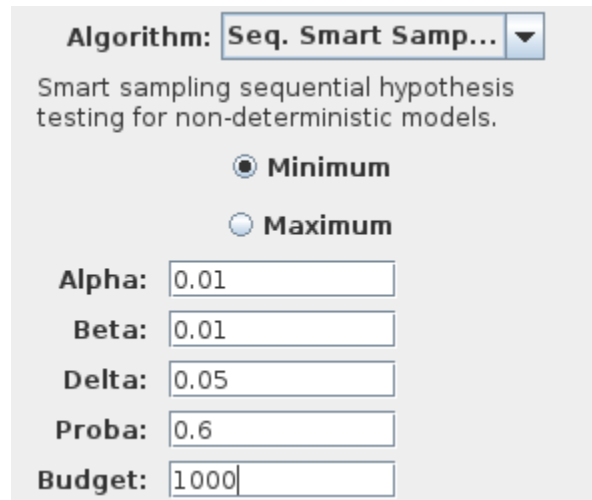
Epsilon:
Delta:
Budget:
Max budget:
Reduction Factor:
Initial probability:
Reward name:
☐ **SPRT**
Threshold:
Alpha:

The iterative steps of the algorithm allocate the simulation budget between each renaming scheduler. The number of scheduler is indeed reduced by at least half (or greater if the factor is greater than 2) at each iteration such that only the bests schedulers are kept. The number of steps (and simulations) is therefore bounded by the simulation budget.

The result of the algorithm outputs the estimated probability of the best scheduler found, as well as the number of steps performed by the algorithm and the total number of simulations.

Smart hypothesis testing algorithm

The algorithm is available in the experimentation tab under the name **Seq. Smart Sampling**. In this algorithm we use the value of the threshold probability to directly calculate the initial allocation of simulation budget. This allows to skip the first step of the analysis. Latter steps are similar to the estimation algorithm in performing a fixed number of simulations for each remaining schedulers, and then keeping the best half of schedulers.



The screenshot shows the configuration interface for the 'Seq. Smart Sampling' algorithm. At the top, there is a dropdown menu labeled 'Algorithm:' with 'Seq. Smart Samp...' selected. Below this, a description reads 'Smart sampling sequential hypothesis testing for non-deterministic models.' There are two radio buttons: 'Minimum' (selected) and 'Maximum'. Below these are five input fields: 'Alpha:' with value '0.01', 'Beta:' with value '0.01', 'Delta:' with value '0.05', 'Proba:' with value '0.6', and 'Budget:' with value '1000'.

If the “average scheduler” or an individual scheduler ever satisfies the hypothesis, the algorithm terminates immediately and reports that the hypothesis is satisfied. If the “best” scheduler ever individually falsifies the hypothesis, the algorithm also terminates and reports the result. Note that this outcome does not imply that there is no scheduler that will satisfy the hypothesis, only that no scheduler was found with the given budget. Finally, if the algorithm refines the initial set of schedulers to a single instance and the hypothesis was neither satisfied nor falsified an inconclusive result is given.

5.4 Reward estimation for MDPs

In addition to estimating minimum and maximum probabilities, PLASMA Lab implements an algorithm to estimate minimum and maximum expected reward, as defined by the rewards properties described below.

5.4.1 Types of reward

Rewards or costs are additional variables assigned to states or transitions. The RML language uses a specific rewards construction described below and on the PRISM website:

```
rewards rewardName

    guard1 : value1;
    [] guard2 : value2;
    [channel] guard3 : value3;

endrewards
```


This construction associates the following values to the reward:

1. The first command defines a state reward that assigns the *value1* to each state of the model that satisfies *guard1*.
2. The second command defines a transition reward that assigns the *value2* to each non synchronized transition that satisfies *guard2*.
3. The third command also defines a transition reward but it assigns the *value3* only to the transitions labeled with *channel* and that satisfy *guard3*.

Three types of reward properties define the value of the reward associated to a path that will be used by the algorithm:

- **Reachability rewards:** The reward is associated to a BLTL property and the value of the reward for a path is the cumulative value (the sum of state rewards and transition rewards) along the path up to the state that satisfies the property (or the last one if the property is not satisfied). Note that this is different from PRISM that only computes reachability rewards for LTL properties that are always satisfied (otherwise the reward is infinite).
- **Cumulative rewards:** The reward of a path is the cumulative reward up to time **T**. This type of reward is easily encoded with PLASMA Lab using a reachability reward with a BLTL property like $G \leq \# \text{ T true}$.
- **Instantaneous rewards:** The reward of a path is the state reward of the state at time **T**. To specify this reward in PLASMA Lab we use the name `rewardName[I]` instead of *rewardName* and a BLTL property like $G \leq \# \text{ T true}$ to encode the timing.

5.4.2 Smart reward estimation algorithm

The algorithm is somehow similar to the probability estimation algorithm. However the set of schedulers is directly initialized at the begin of the algorithm using all the simulation budget, and the first iteration performs one simulation for each scheduler. For each scheduler the algorithm evaluates the expected reward at the time the BLTL property is satisfied. Then after each iteration the best schedulers are kept and the number of simulations is increased according to the reduction factor.

Algorithm: Smart sampling ▼


Smart sampling Monte Carlo estimation algorithm for non-deterministic models.

☒ **Minimum**
☐ **Maximum**


Epsilon:
Delta:
Budget:
Max budget:
Reduction Factor:
Initial probability:
Reward name:
☒ **SPRT**
Threshold:
Alpha:

For reachability rewards an additional computation is performed. Since the algorithm only consider bounded properties, we evaluate the hypothesis that the probability to satisfy the BLTL is at least greater than a given **Threshold**. This is computed using an **SPRT** test with a confidence parameter **Alpha**. If the test is satisfied the result will display the property indicator in green. Otherwise it stays red.

Hypothesis accepted by the SPRT test:

Experimentation results		plot			
#		Name	Min of "rounds"	Iterations	# Simulations
1		FDone	1.3878326996197718	10	99930

Hypothesis rejected by the SPRT test:

Experimentation results		plot			
#		Name	Min of "rounds"	Iterations	# Simulations
1		FDone	1.2129277566539924	10	99930

5.5 Importance splitting

This algorithm allows to estimate the probability of rare events (with very low probability) more efficiently than the classical Chernoff-Bound. The idea is to decompose the verification of property ϕ into several properties such that $\phi_1 \Rightarrow \phi_2 \Rightarrow \phi_3 \Rightarrow \dots \Rightarrow \phi_n = \phi$. These properties defined a set of levels that a trace need to satisfy before ϕ is satisfied.

5.5.1 RML observers

The importance splitting algorithm only works with a special type of requirement from the observer plugin included in the RML plugin. An observer requirement encodes the automata that determines the value of a BLTL property along a trace. If this property can be decomposed the automata computes a score that determines which level has been reached by the trace.

Syntax

An observer is similar to a module from the RML language with the following particularities:

- An observer is declared with the construction `observer ... endobserver` in place of `module ... endmodule`.
- An observer can declare local variables.
- An observer can reference in its expressions external variables from the model.
- An observer can only have non synchronized commands.

An observer requirement can have several observers, as well as constants, formulas, and global variables. It must contain one variable **score** and one variable **decided**.

As example this observer is used in the simple chain model:

```
observer chainObserver
  score : double init 0;
  decided : bool init false;
```

(continues on next page)

(continued from previous page)

```

[] s != 11 & score < s -> (score'=score+1);
[] s>=10 -> (decided'=true);
endobserver

```

Semantics

The observers are executed sequentially after each simulation step of the model:

- Each observer in the requirement is executed once in sequential order from top to bottom of the requirement file.
- Each enabled command from an observer is executed once, again sequentially in the order of declaration from top to bottom.
- The observers and the model are simulated as long as the variable decided is false. Therefore the observers must encode a temporal bound and update the variable decided once the temporal bound is exceeded.
- The variable score is used by the algorithm to determine which level the current trace has reached.

From BLTL to observers

PLASMA Lab include a tool that can automatically translate a BLTL property into an observer requirement. This functionality is available with a right click on the name of the BLTL property in the *Project Explorer*. The property can have limited nesting of temporal operators. It produces an observer requirement with several observers (one for each temporal operator). The user must then edit this requirement to compute an adequate score.

5.5.2 Importance Splitting Algorithms

The principle is to specify a fixed budget that is the number of traces executed at each level. The traces that reached the next level are kept. The ones that failed are restarted from a successful trace. The algorithms then compute a series of conditional probabilities to reach a level n from a level $n+1$ and the final result is the product of these conditional probabilities.

Two different importance splitting algorithms are implemented in PLASMA Lab:

- **Fixed levels algorithm:** The levels are specified by the user that inputs a series of decimal values separated by spaces. These values correspond to the score of the different levels. The algorithm simulates the traces up to a state that satisfies the next level or up to the time bound if the trace is not successful. The last state of the successful traces correspond to the initial states for the simulations up to the next level.

The screenshot shows a configuration window for the 'Importance splitting algorithm'. At the top, a dropdown menu is set to 'Importance splitt...'. Below it, a text label reads 'Importance splitting algorithm for rare event estimation.' There are two radio buttons: 'Adaptive algorithm' (which is unselected) and 'Fixed levels algorithm' (which is selected). A 'Budget:' label is followed by a text input field containing '1000'. A 'Maximum score:' label is followed by an empty text input field. A 'Levels:' label is followed by a text input field containing '2.5 5 7.5 10'.

- **Adaptive algorithm:** The number of levels is not known a priori. The user input the maximum score of the trace, that is reached when the trace satisfies the BLTL property encoded by the observers. The algorithm simulate all the traces up to the time bound. The traces with the minimum score are restarted at the next next using an initial state of a “better” trace. This maximizes the number of levels and improve the variance of the estimator.

The screenshot shows a configuration window for the 'Importance splitting algorithm for rare event estimation'. It features a dropdown menu for 'Algorithm' set to 'Importance splitt...'. Below this, there is a 'Budget' input field with the value '1000'. Two radio buttons are present: 'Adaptive algorithm' (which is selected) and 'Fixed levels algorithm'. The 'Adaptive algorithm' section includes a 'Maximum score' input field with the value '10'. The 'Fixed levels algorithm' section includes a 'Levels' input field which is currently empty.

5.6 Importance sampling

Importance sampling is another technique to improve rare events simulation. It works by using an alternate probability distribution in the model, such that rare properties will appear more often in the simulations. SMC algorithms take into account the likelihood ratio of the traces generated with this new probability distribution in order to compute an unbiased result.

5.6.1 RML with importance sampling

From version 1.3.4 of PLASMA Lab allows to add sampling parameters in the RML language.

Model type

Importance sampling uses a different simulator in order to simulate the model according to the sampling parameters and to compute the modified rate of each run. This simulator is enabled by using the keyword *sampling* after the model type. Therefore the following model types are available for importance sampling:

- dtmc sampling
- ctmc sampling
- mdp sampling
- mdp shd sampling
- mdp sml sampling

Sampling parameters

Sampling parameters are declared with { *expr* } before the normal rate of an action. This new rate multiply the normal rate of the action.

Algorithms

Importance sampling is implemented with the following algorithms:

- Monte Carlo
- Chernoff

- Chernoff ND
- Cross entropy

5.6.2 Cross entropy algorithm

The choice of a sampling distribution is critical to get a good estimate with the importance sampling technique. The cross entropy technique allows to discover an optimal distribution using an iterative process:

- Starting from an initial distribution, the algorithm computes new values for the parameters by counting the number of times each command is used in successful traces.
- The algorithm performs this process during several iterations, until the parameters have reached a sufficient precision.

PLASMA Lab cross entropy algorithm assumes that there exists three special categories of variables in the model:

- A set of variables that corresponds to the sampling parameters. They are named **lambda1**, **lambda2**, **lambda3**, etc.
- A set of variables that counts the occurrence of a transition associated to a sampling parameter. They are named **nb_lambda1**, **nb_lambda2**, **nb_lambda3**, etc.
- A set of variables that defines the normal rate of the transitions. They are named “**rate_lambda1**”, “**rate_lambda2**”, “**rate_lambda3**”, etc.

Currently only the RML language enable importance sampling. To use cross entropy with RML in PLASMA Lab the user first needs to edit his model such that:

- Each action is associated to a sampling parameter named **lambda1**, **lambda2**, **lambda3**, etc.
- Sampling parameters are declared globally as constant, possibly with an initial value.
- For each sampling parameter, there exists a count variable declared globally as an integer, initialized at 0, and named **nb_lambda1**, **nb_lambda2**, **nb_lambda3**, etc.
- Each action increments its count variable if it is taken. For instance the following command:

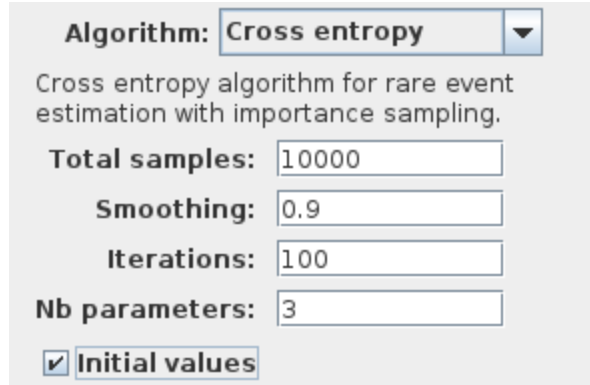
```
[] state1 >= 2 -> mu : (state1'=0);
```

is transformed with sampling parameters into:

```
[] state1 >= 2 -> {lambda2} mu : (state1'=0) & (nb_lambda2'=nb_lambda2+1);
```

- For each sampling parameter there exists a label named “**rate_lambda1**”, “**rate_lambda2**”, “**rate_lambda3**”, etc, that defines the normal rate of the action. For instance, in the above command, the rate of the action is label “**rate_lambda2**” = mu;

PLASMA Lab implements the cross entropy algorithm with the following parameters:



The screenshot shows a configuration window for the 'Cross entropy' algorithm. It includes a description: 'Cross entropy algorithm for rare event estimation with importance sampling.' Below this are four input fields: 'Total samples' set to 10000, 'Smoothing' set to 0.9, 'Iterations' set to 100, and 'Nb parameters' set to 3. At the bottom, there is a checked checkbox labeled 'Initial values'.

- **Total samples** is the number of simulations at each iteration.
- **Smoothing factor** is a parameter between [0,1] use to decrease the probability of a command the has never been used in the previous iteration. This is less abrupt than setting the sampling parameter to zero.
- **Maximum iterations** is the number of iterations of cross entropy to perform.
- **Number of parameters** is the number of sampling parameters in the model.
- **Initial values** is an option that uses a uniform simulator at the first iteration of the algorithm in order to guess the initial distribution of the parameters.

5.7 Checking unbounded A-BLTL properties

PLASMA Lab can check unbounded properties from the A-BLTL logic. The algorithm that is used involved a reachability check of the underlying finite automata model. To do this PLASMA Lab calls the PRISM model-checker. We explain below how to configure PLASMA Lab in order to use this functionality:

- Step 1: install PRISM in your system.
- Step 2: download the script [prism_reach](#).
- Step 3: in the script edit the line with the path to PRISM
- Step 4: launch PLASMA Lab with the option `-Dprism_reach=path` where *path* is the path to the script.

DEVELOPER MANUAL

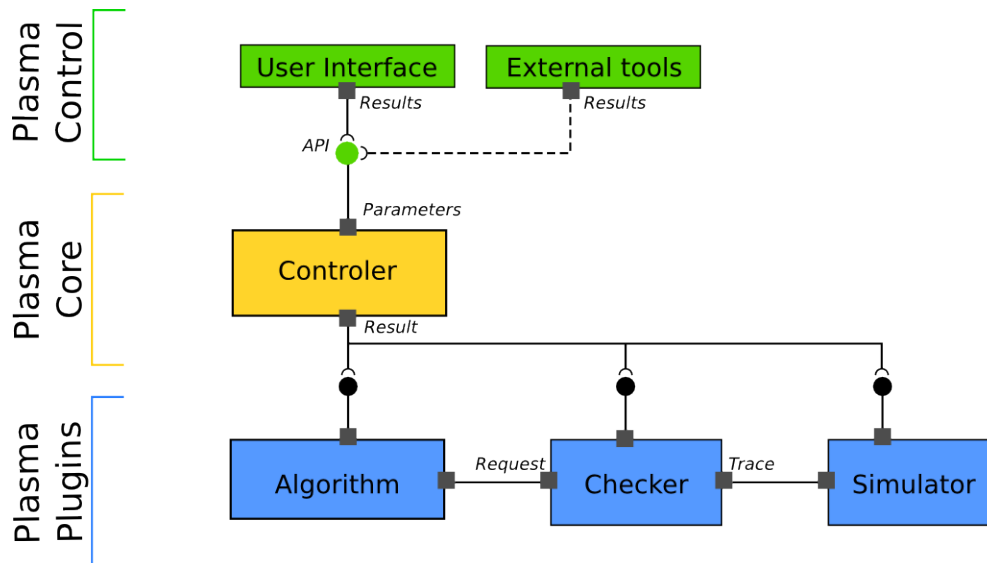
This chapter is addressed to the developer who wants to contribute to PLASMA Lab or use it in their own project. First, we present an overview of the architecture, describing the basic concepts inside PLASMA Lab. Each main components/concepts of the platform is detailed in a separate section. Finally, we propose a short tutorial that describes how to implement new features in the tool.

The Javadoc of PLASMA Lab's API can be found [here](#) (select your version).

6.1 Architecture

One of the main differences between PLASMA Lab and other SMC tools is that PLASMA Lab proposes an API abstraction of the concepts of stochastic model simulator, property checker (monitoring) and SMC algorithm. In other words, the tool has been designed to be capable of using external simulators or input languages. This not only reduces the effort of integrating new algorithms, but also allows us to create direct plug-in interfaces with standard modelling and simulation tools used by industry. The latter being done without using extra compilers

The figure below presents the tool architecture: - The core of Plasma Lab is a lightweight **Controller** that manages the experiments and the distribution mechanism. - It implements an API to control the experiments either through **user interfaces** or **external tools**. - It loads three types of plugins: 1. **Algorithms**, 2. **Checkers**, and 3. **Simulators**. These plugins communicate with each other and with the controller through the API.



6.1.1 MVC Pattern

PLASMA Lab can also be seen as implementing the Model-View-Controller (MVC) pattern. Here, *View components* use the Controller API to communicate with the *Controller component*. In turn, the *Controller* controls the *Model components*.

In the following section we detail which PLASMA Lab project corresponds to which type of components and what is their purpose.

View Components

- **PLASMA Lab Graphical User Interface** – *fr.inria.plasmalab.ui*

This is the GUI of PLASMA Lab, implemented with Swing. It incorporates all functionalities of PLASMA Lab and allows to open and edit PLASMA files (project files).

- **PLASMA Lab Command Line** – *fr.inria.plasmalab.terminal*

This view is a command line interface for PLASMA Lab. It provides experimentation and simulation functionalities, including the command line for PLASMA Lab service.

- **PLASMA Lab Service** – *fr.inria.plasmalab.service*

This is a small graphical interface for PLASMA Lab distributed service. It is deployed on a distant computer to help run a distributed experiment.

- **PLASMA2Simulink** – *fr.inria.plasmalab.matlab_ui*

This is a small graphical “APP” running from Matlab to use PLASMA Lab SMC algorithms with a Simulink model.

Model Components

The Model part contains three type of components. **Algorithm**, **Simulator** (Model) and **Checker** (Requirement/Property). They all are related by a simple workflow. An Algorithm asks a Checker to check a property. The Checker will in turn ask the Simulator to run a simulation and will use the trace returned to decide the property. The decision will be returned to the Algorithm.

Checker and Simulator components are **plugins** that can be loaded on startup. Algorithms were designed to be plugins too but this feature was discarded. The plugin system use the Java Simple Plugin Framework (JSPF).

Here is a list of project for each of these categories.

Algorithms

- *Monte Carlo based algorithm* – *fr.inria.plasmalab.montecarlo*
It includes algorithms for nondeterministic models.
- *Sequential Hypothesis Testing* – *fr.inria.plasmalab.sequential*
It includes the sequential algorithm for nondeterministic models
- *Importance splitting algorithms* – *fr.inria.plasmalab.importancesplitting*
- *Importance sampling and the cross entropy algorithm* – *fr.inria.plasmalab.importancesampling*
- *CUSUM* – *fr.inria.plasmalab.cusum*

Checker

- *Bounded Linear Temporal Logic* – *fr.inria.plasmalab.bltl*

- *Adaptive Linear Temporal Logic* – *fr.inria.plasmalab.altl*
- *Global Contract Specification Language* – *fr.inria.plasmalab.gcsl*
- *Nested BLTL* – *fr.inria.plasmalab.nested*

Simulator

- *Reactive Module Language* – *fr.inria.plasmalab.rmlbis*

The project also includes the simulator for *Adaptive RML* and the observers requirements used with importance splitting.

- *Biological language* – *fr.inria.plasmalab.bio*
 - *MatLab interface* – *fr.inria.plasmalab.matlab*
 - *SystemC interface* – *fr.inria.plasmalab.systemc*
-

Controller Component

The Controller component (project *fr.inria.plasmalab.controller*) acts as an interface between the Models and the Views. The second purpose of the Controller is to initialize PLASMA Lab with configuration files and load plugins.

Note: The project GUI has its own Controller object to deal with file management, interface callback, etc.

Common

Various projects define classes and interfaces used by several component. The main one is the Workflow (project *fr.inria.plasmalab.workflow*) that contains definitions of several interfaces and classes, as well as a set of PLASMA Lab Exceptions.

Here is a list of common project

- **Workflow** – *fr.inria.plasmalab.workflow*
Interface and class definitions used by several components, in particular Data and Factory interfaces for Simulator and Checker components. Definitions of Plasma Lab exception classes.
- **Algorithm** – *fr.inria.plasmalab.algorithm*
Defines Data and Factory interfaces for Algorithm components.
- **Distributed** – *fr.inria.plasmalab.distributed*
Defines the Restlet architecture and components used in all distributed Algorithms as well as a Heartbeat mechanism to detect loss of connection from Workers.
- **GUI command line** – *fr.inria.plasmalab.uiterminal*
It handles the launch and the configuration of the GUI from a terminal.
- **Common command line** – *fr.inria.plasmalab.common-cli*
It provides common features between the two command line terminal projects.
- **Text data** – *fr.inria.plasmalab.text_data*
This loads data as plain text, without syntax. It is used in particular to load a project whose plugin is missing and incorrect.

- **Root** – *fr.inria.plasmalab.root*

This project contains no code but is the base project for compiling PLASMA Lab.

6.2 Plugins

There are three types of plugins, Simulator, Checker and Algorithms. A plugin is build around two main classes: a Factory, and a Model/Requirement/Scheduler class.

The *Factory* can be seen as the plugin entry point. It implements methods to retrieve the plugin's name and description. But its main task is to instantiate the class that is implementing the Simulator, the Checker or the Algorithm interface.

Models and requirements factories inherit from the *AbstractModelFactory* or *AbstractRequirementFactory* classes, that both implement the *AbstractDataFactory* interface. Algorithms factories inherit the *InterfaceAlgorithmFactory*.

PLASMA Lab plugin system use the **Java Simple Plugin Framework**. To load a plugin, the factory class must implement the *Plugin* interface and have the tag *@PluginImplementation* before the class declaration. More details and a 5 min tutorial can be found on their website:

<http://code.google.com/p/jspf/>

6.3 Simulator

A simulator executes a model to generate observation traces. In PLASMA Lab a model is described in a textual content. This can either contain the description of a model or a link to an external simulator. In that latter case, the simulator plugin loaded by PLASMA Lab acts only as an interface between PLASMA LAB and the external simulator.

6.3.1 AbstractModel

Each simulator inherits from the *AbstractModel* class that itself inherits from *AbstractData*. *AbstractData* describes a container for a data such as a model (but also a requirement) and provides functions such as *getName*, *getContent* and *checkForErrors*. As this abstract class is quite simple we won't get too much into details and we focus on *AbstractModel*.

CheckForErrors

```
public boolean checkForErrors();
```

checkForErrors comes from the *AbstractData* class. A call to this function needs to parse the model content and to initialize any data structure needed before starting a simulation. If errors were found, the function returns true.

After modifying the model, a single call to *checkForErrors* is needed before starting a batch of simulation (several calls to *newPath*).

New Path

```
public InterfaceState newPath() throws PlasmaSimulatorException;
```

```
public InterfaceState newPath(List<InterfaceState> initTrace) throws
↳ PlasmaSimulatorException;
```

```
public InterfaceState newPath(long seed) throws PlasmaSimulatorException;
```

The **newPath** methods initialize a new trace and return the initial state. A version of this method takes an initial trace as a parameter to start the simulation from the last state of this trace. another version takes a seed as a parameter. The seed can then be freely use by the simulator. Depending on the semantic used in each simulator, the seed may not have the same usage from one simulator to another. The method may throw a *PlasmaSimulatorException* if an error occurred in the simulator.

Simulation

```
public InterfaceState simulate() throws PlasmaSimulatorException;
```

Once the trace was started with *newPath*, the **simulate** method is used to progress the simulation step by step. A call to *simulate* will thus execute a single simulation step. The *simulate* method returns the new state added to the trace (that is to say the last state of the trace). If a deadlock is reached (no new state can be added), a *PlasmaDeadlockException* is thrown. The method may throw a *PlasmaSimulatorException* if an error occurred in the simulator.

```
public void backtrack() throws PlasmaSimulatorException;
```

The **backtrack** method cancels the last call to *simulate*.

```
public void cut(int stateNb) throws PlasmaSimulatorException;
```

The **cut** function takes an integer *stateNb* as a parameter and cut the current trace at the *stateNb* position. Every state before this point is deleted as if the state at position *stateNb* was the initial state.

```
public void clean() throws PlasmaSimulatorException;
```

The **clean** method is called after a simulation was completed and before a new one start. It is used in case some operations must be done in order to return to a safe state. It must not necessarily be implemented by the simulator. *AbstractModel* already provides an implementation of this method that does nothing.

State and trace getters

AbstractModel provides several getters to states.

```
public InterfaceState getCurrentState();
```

Retrieve the head of the current path, ie: the latest state generated.

```
public InterfaceState getStateAtPos(int pos);
```

Retrieve the state at the position given in parameter.

```
public List<InterfaceState> getTrace();
```

Retrieves the current trace. A simulation **trace** (or path) is a list of states.

```
public int getTraceLength() {  
    return getTrace().size();  
}
```

This method is already implemented by *AbstractModel* and return the length of the trace (number of states).

```
public abstract int getDeadlockPos();
```

This method return the position of the last state in the trace if a deadlock occurred or -1 if no deadlock occurred.

Identifiers getters

Several methods allows to access the identifiers used by the model. These identifiers are used to communicate values to the checker or to the user interfaces.

```
public abstract Map<String, InterfaceIdentifier> getIdentifiers();
```

This method returns a map of all the identifiers that can be evaluated on a state and used in the requirements.

```
public abstract InterfaceIdentifier[] getHeaders();
```

This method returns an array of identifiers that are followed along a trace in simulation mode. They will appear for instance in the simulation results panel of GUI in the same order as in the array.

```
public abstract InterfaceIdentifier getTimeId();
```

This method returns the identifier that counts the continuous time. It may be null if the model has no continuous time.

```
public abstract boolean hasTime();
```

This method returns true if the model provides continuous time (like CTMC).

```
public abstract List<InterfaceIdentifier> getStateProperties();
```

This method return a list of identifiers that represent state properties. A state property is a boolean formulae that is evaluated on a state. It is represented by an *InterfaceIdentifier* whose value will be obtained from a state. It is used in Simulation mode in the properties panel.

Other methods

```
public void setValueOf(Map<InterfaceIdentifier, Double> update) throws  
↳ PlasmaSimulatorException;
```

This method allows to modify the initial value of the model. It is used to perform optimization of the model parameters or to start the simulator in a different state, for instance by the importance splitting algorithm.

```
public List<Variable> getOptimizationVariables();
```

This method returns a list of variables on which to perform optimization. Implemented by default in *Abstractmodel* to return an empty list.

```
public List<VariableConstraint> getOptimizationConstraints();
```

This method returns a list of constraint on the optimization variables. Implemented by default in *Abstractmodel* to return an empty list.

6.3.2 InterfaceIdentifiers

InterfaceIdentifier is an interface used in PLASMA Lab to represents an object acting as an identifier.

A **GenericIdentifier** implementation of the interface is provided in the workflow project. It has only a name attribute.

6.3.3 InterfaceState

InterfaceState represents a mapping from *InterfaceIdentifiers* to values that constitutes a single state of a simulation. It can also reference an *InterfaceTransition* object that represents the transition from which this state was reached. It extends the *ResultInterface*.

The interface provides two methods for accessing the values of the state, either through an *InterfaceIdentifier* object or through the name of the identifier.

```
public Double getValueOf(InterfaceIdentifier id) throws PlasmaSimulatorException;
```

```
public Double getValueOf(String id) throws PlasmaSimulatorException;
```

A **GenericState** implementation of the interface is provided in the workflow project.

6.3.4 Optimization variables

Optimization variables are used to initialize a model with a range of initial state in the same experiment. PLASMA Lab uses this feature with the *optimization procedure*.

Optimization variables can be declared in the model or in the requirements and are retrieved using the **getOptimizationVariables** function.

6.4 Requirement checker

A requirement (or property) is a specification that is observed other a trace. It is generally specified in a formal logic like the *Bounded Linear Temporal Logic* and comes as a textual description. The checker implements the algorithms necessary to checked a trace according to the semantics of the requirement.

6.4.1 AbstractRequirement

Each checker inherits from the *AbstractRequirement* class that itself inherits from *AbstractData*.

CheckForErrors

```
public boolean checkForErrors();
```

checkForErrors come from the *AbstractData* class. A call to this function will parse the requirement and initialize any data structure needed before checking the requirement. If errors were found, the function returns true.

After modifying the requirement, a single call to *checkForErrors* is needed before starting a batch of verification (several calls to *check*).

Check

```
public Double check(InterfaceState path) throws PlasmaCheckerException;
```

The **check** function takes an initial *InterfaceState* as a parameter and checks the requirement from this state. It returns a double value once a decision has been made. Depending on the requirement language, the number returned may represent a Boolean or a decimal value.

Although the simulation is initialized (call to *newPath*) before starting the verification, the checker is in control of the simulation. During the simulation, the checker will indeed call the *simulate* method to add new states to the trace on demand.

```
public Double check(int untilStep, InterfaceState path) throws PlasmaCheckerException;
```

A second version of the *check* function takes an integer as an additional parameter. This parameter, *untilStep*, tells the checker to verify the simulation until a given step. This function is used with the simulation mode of PLASMA Lab. Using this mode, the simulation is directly under control of the user. If the verification reach the *untilStep* bound, the Checker must not call the *simulate* method. If no decision can be made once the bound has been reached, the Checker can use the *Double.NaN* constant (Not A Number) with the meaning *undecided*.

```
public abstract Double check(String id, double untilValue, InterfaceState path) throws PlasmaCheckerException;
```

This third version of the *check* function additionally considers as parameters the name of an identifier and a final value. It should check the requirement until the given variable reaches a certain value or a deadlock is reached. However, this method doesn't guarantee termination in case the goal value is never reached.

Others

```
public void clean();
```

The **clean** method is called after an experiment was completed and before a new one start. It is used in case some operations must be done in order to return to a safe state.

```
public void setModel(AbstractModel model);
```

Set the *AbstractModel* object to verify.

```
public List<InterfaceState> getLastTrace();
```

Retrieve the last trace that has been checked with the checker.

```
public void setSpecificParameters(Object[] parameters) throws PlasmaParameterException;
```

Optional method whose sole purpose is to pass specific parameters for some requirements types.

```
public List<Variable> getOptimizationVariables();
```

Retrieve optimization variables declared by this requirement.

```
public List<AbstractRequirement> generateInstantiatedRequirement();
```

Instantiate a list of *AbstractRequirement* from this requirement and the range of *requirement variables* declared in it. *Requirement variables* are similar to *optimization variables* but are used to generate a set of requirements instead of a set of initial states.

6.5 Algorithms

In this section we introduce a generic view of the algorithm class in PLASMA Lab. For this purpose we consider the case of a local (non distributed) algorithm. The implementation of distributed algorithms is presented in the next *section*.

6.5.1 InterfaceAlgorithmFactory

As presented before, algorithms factories implements the *Plugin* interface. They are responsible for parsing the algorithm's parameters and constructing the scheduler class of the algorithm.

As each algorithm needs various parameter as input, we developed generic objects to declare these parameters. Each algorithm factory declares a list of *SMCParameter* objects. This list is then used by PLASMA Lab to retrieve correct values from the user.

First we present each type of parameters. Then we describe the way a list of parameter is used and values retrieved by PLASMA Lab.

SMCParameter

The *SMCParameter* class is the parent class of all parameters. Its basic attributes are a *name*, a *tip* and a Boolean that defines if the parameter represents a Boolean value or not.

Used alone, an *SMCParameter* represents a simple text, double or Boolean value.

SMCOption

A *SMCOption* is a Boolean parameter that contains a list of other *SMCParameter*. If the *SMCOption* is set to true, its children parameters are enabled and can be filled by the user.

SMCOption behaves like a check box object, and it is represented by one in PLASMA Lab GUI.

SMCAalternatives

An *SMCAAlternative* is a Boolean parameter that contains a list of other *SMCParameter* and points to another *SMCAAlternative* (like the linked list mechanism). If the *SMCAAlternative* is set to true, its children parameters are enabled and can be filled, while every other *SMCAAlternative* in the linked list are set to false.

An *SMCAAlternative* behaves like a radio button object, and it is represented by one in PLASMA Lab GUI.

SMCObjectParameter

An *SMCObjectParameter* defines a parameter to be an *AbstractModel*. It is specifically used in the CUSUM algorithm, that needs to be executed on two different models (the basic GUI can only select one model).

The *SMCObjectParameter* is interpreted by the PLASMA Lab GUI as a ComboBox containing models from the currently selected project.

Parameters workflow

- **Creating parameter widgets**

An *InterfaceAlgorithmFactory* object defines a list of *SMCParameter* that correspond to their algorithm. This list is accessed by the *ParametersPanel* class using the **getParametersList** method. A widget (CheckBox, RadioButton, TextField, ...) is created in the GUI for each parameter, including children parameters. Some widgets depend on their parent widget to be enabled.

- **Retrieving parameters values**

When launching a new experiment, the *ExperimentPanel* access to the *ParametersPanel* corresponding to the selected algorithm and retrieves a map of String/Object pairs. For each pair, the String corresponds to the name of the parameter (as provided by the *InterfaceAlgorithmFactory* and displayed on the widget) and the Object is the value of the parameter. As said previously, the value could be of type Boolean, String, or even *AbstractModel*.

Additional parameters may be added to the map such as *distributed*, which informs the system if a distributed version of an algorithm should be used.

The map and the selected *InterfaceAlgorithmFactory* is then transmitted to the *ExperimentationManager*. The *ExperimentationManager* calls the **createScheduler** function on the factory with the parameters map as an argument. This function parses the values of the parameters and it checks if they suit the algorithm. It then creates an instance of the *InterfaceAlgorithmScheduler* ready to be executed and returns it.

- **Alternative workflows**

The previous method to retrieve parameters values requires the use of widgets, which may be unavailable with user interfaces different than the GUI (command line, interface with another tool, ...).

An alternative workflow (used by the command line) is to construct directly a map of parameters name and values, defined by the user. In that case the user must be aware of which parameters and values are valid and necessary. The map is then transmitted to the *createScheduler* function.

Another solution is to call the *fillParametersMap* method of *InterfaceAlgorithmFactory*. This method converts an array of parameters values (written as String values) into a map of String/Object pairs. This method fills the map with the parameters value taken from the array by converting the values to their correct types. The content and the order of the parameters in the String array is however entirely dependant of the *InterfaceAlgorithmFactory* implementation.

6.5.2 InterfaceAlgorithmScheduler

```
public interface InterfaceAlgorithmScheduler extends Runnable
```

This class provides the main interface for an SMC algorithm (both local or distributed). This interface extends the *Runnable* interface. Thus an algorithm will have a *run* method.

```
public void setExpListener(ExperimentationListener listener);
```

The *InterfaceAlgorithmScheduler* also defines a *setExpListener* method. Object implementing the *ExperimentationListener* interface will be kept up to date on the experiment progress using calls to functions such as *notifyAlgorithmStarted*, *notifyAlgorithmCompleted* and *publishResults*.

```
public void abortScheduling();
```

The *abortScheduling* method is called when the user wants to stop a running experiment before it finishes.

Run method of an SMC Algorithm

The run method of an SMC Algorithm is where the work is done. For instance the run method of the basic Monte Carlo algorithm could be based on the following pseudo-code:

```
initialize(model, property)
listeners.notifyAlgorithmStarted

while(continue)
    state = model.newPath
    positive += property.check(state)
    total ++
    if(total > required)
        continue = false

listeners.publishResults(getResults(positive))
listeners.notifyAlgorithmCompleted
```

For each run, the model initializes a new trace. The property is checked starting from the initial state. If the property has been checked on enough traces, the algorithm terminates and the results are published.

6.5.3 AbstractAlgorithm

This class provides a generic implementation of the basic methods for an SMC algorithm. It provides in particular an *initializeAlgorithm* method that should be called at the beginning of the *run* method. This method initializes optimization attributes of the algorithm.

6.5.4 SMCRresult

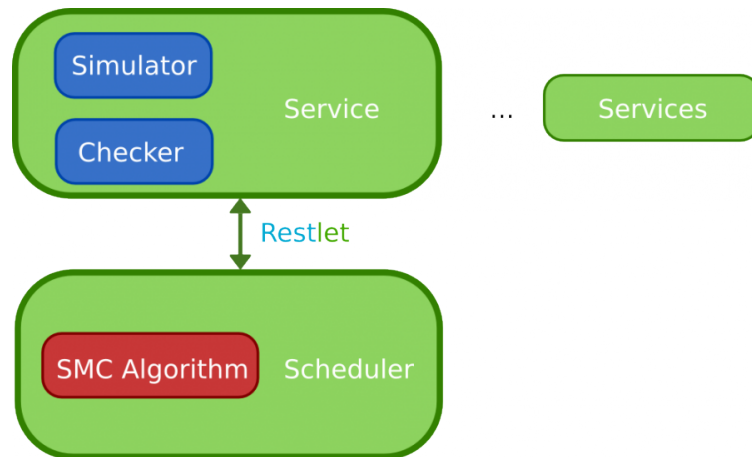
The data structure used to return results of an experimentation is the *SMCRresult* interface. This interface inherits from *ResultInterface*.

In PLASMA Lab, a class implementing the *ResultInterface* represents the output of a task such as a simulation or an experiment. It provides access to a header array of type String with each header being associated to a value.

The *SMCRresult* interface adds a specific *getPr* method to get a single result of the experiment. This result is a *double* value, that represents for instance a probability or a Boolean value with 1 meaning true and 0 meaning false. Other results are accessed by the *getValue* method of *ResultInterface*.

6.6 Distributed algorithms

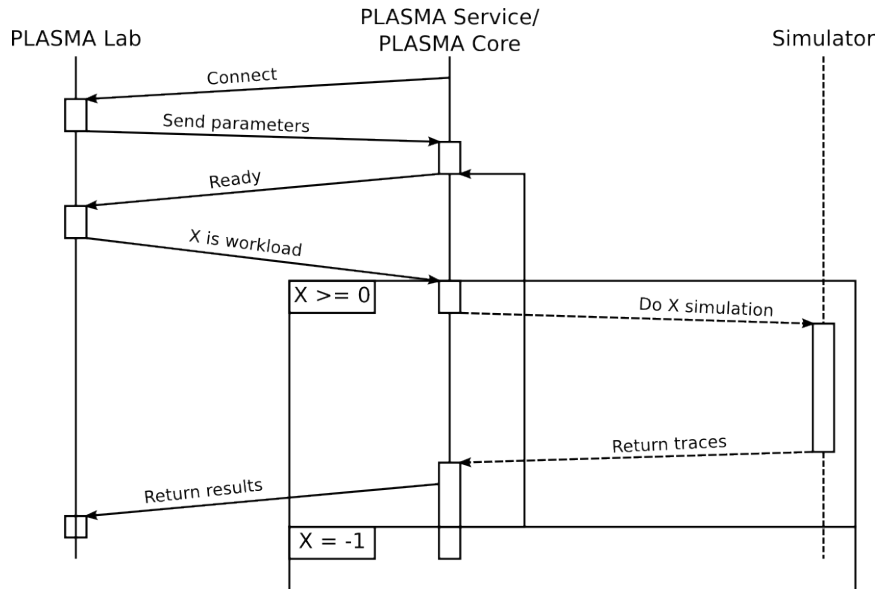
PLASMA Lab can run experiments in a distributed mode. This mode uses a second software, called **PLASMA Lab Service**, that is a small PLASMA Lab client deployed on a distant computer.



In distributed mode the *Simulator* and the *Checker* components are deported to the *Service*, while the *Algorithm* is split between a **Scheduler**, running in PLASMA Lab, and a **Worker** running in the *Service*. The communication between the *Scheduler* and the *Worker* uses **RestLet**.

6.6.1 Sequence diagram

Sequence diagram of a session between PLASMA Lab and PLASMA Service:



Step 0: Publishing a Restlet service

PLASMA Lab: I need help to run this experiment.

When launching a distributed experimentation, PLASMA Lab publishes an interface on a Restlet Server (this server being part of the PLASMA Lab code). Publishing this interface will allow other programs to retrieve the interface and use its functions.

We call Client a PLASMA Service instance and the Server the PLASMA Lab instance who published the interface.

Step 1: Connecting to PLASMA Lab

PLASMA Service: I can help you.

PLASMA Lab: Ok, here are the experiment parameters.

When a client connects itself to the server, it retrieves the RMI Interface and it calls the connect function. This function returns the experiment parameters containing the model and the list of requirements to check. These parameters also contains an ID to allow clients to identify themselves.

Step 2: Getting a share of the work

PLASMA Service: I am ready to work.

PLASMA Lab: Run K simulation and check the properties.

The client then calls the service ready function. This function will inform the server that this client is ready and will return an integer K. This integer represents the number of traces that the client has to generate. K can take several values:

- K = -1 means that the work is done, the client can disconnect.
- K = 0 tells the client to wait. This is used to wait for more client to connect before launching the simulation.

- $K > 0$ tells the client to generate K execution traces and check the properties on these traces.

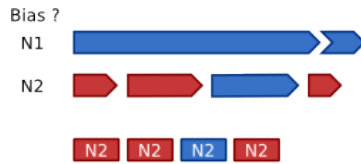
Step 3: Returning the results

PLASMA Service: Here are my results.

The client calls the work done function. This function sends the results to the server. The client then return to step 2.

6.6.2 Scheduler

Avoiding bias: Depending on the requirement, a negative trace – *on which a property does not hold* – can be faster to check than a positive trace – *you have to read the whole trace to see that it is a positive trace* –, or the contrary. In both cases, a bias can form on a distributed SMC algorithm.



Scheduler ensure equity



Our scheduler keeps the order of each task assigned to a client and only takes results into account on this order. This ensure that a task launched at *time t* will be taken into account before any task launched at a *time > t*.

Moreover, this scheduler ensures that a faster client will be given more work and contribute more to the overall effort. We have also implemented some ideas coming from the Slow-Start algorithm used in TCP to reduce the client-server communications.

A *Scheduler* implements the same interface as a local algorithm:

- `fr.inria.plasmalab.algorithm.InterfaceAlgorithmScheduler`

The `fr.inria.plasmalab.distributed.algorithm.AbstractScheduler` class provides a generic implementation of several methods used to create a distributed algorithm:

- The `setServices` method set the parameters for the distributed experiment (port, number of threads, size of the batch, factories).
- The `postErrorMessage` method posts an error message to the experimentation manager and abort experimentation.
- The `putResult` method adds a result to the *TaskScheduler*.
- The `registerNewNode` method registers a new node on the *NodeTaskManager*.
- The `schedule` method sends an order to a node requesting one. This can either be a BATCH order, a WAIT order or a TERMINATE order.
- The `initializeServer` method performs a generic initialization of the scheduler. It should be called at the beginning of the *run* method.
- The `getNextResult` method waits for a result and removes it from the nodeTaskManager. If a stop order is received the method returns null.
- The `reassignTaskTo` method reassigns an unassigned task to a node. If the unassigned task points to a null order, or if there is no assigned task, it returns null.

6.6.3 Worker

Each of our distributed implementation are based on a similar protocol described in the earlier sequence diagram. However this protocol only depends on the Scheduler-Worker implementation and could be changed depending on particular needs. The only constraints being that the **connect** method of the *Worker* will be called to initiate the communication with the *Scheduler*.

A *Worker* implements:

- *fr.inria.plasmalab.algorithm.InterfaceAlgorithmWorker*

The *fr.inria.plasmalab.algorithm.InterfaceAlgorithmWorker.AbstractWorker* class provides a generic implementation of several methods used by workers:

- The *processTerminateOrder* method processes a TERMINATE order (by doing nothing).
- The *processWaitOrder* method processes a WAIT order: it gets the waiting time parameters from the order and it then waits for this duration.
- The *start* method starts the worker and processes the orders.

6.6.4 Order

Whenever a *Worker* tells the *Scheduler* it is ready to work, the *Scheduler* will transmit an **Order**. This *Order* indicates for instance the number of simulation – **a batch** – to run and check. Other types of *Order* can be transmitted:

- **Batch** - Indicate the number of simulations to run and check
- **Wait** - Indicate to wait for a given time
- **Terminate** - Indicate to disconnect and close the Service

6.6.5 Task Manager

Each *Order* is associated to a unique identifier, a **Task**. A *Task* will then be assigned to a *Worker*. These associations are handled by the **NodeTaskManager**.

As explained in the bias section, results must be taken into account in the same order the corresponding task were assigned. This is done by the **TaskScheduler**.

6.6.6 Heartbeat

In case a *Worker* disconnects, or another error happens, the *Task* is re-affected by the *NodeTaskManager* to the next available *Worker*.

A **Heartbeat** system allows the *NodeTaskManager* to monitor the liveness of *Worker*. Using a UDP socket, *Worker* must regularly transmit heartbeat packets. If these transmissions cease, *Worker* are declared disconnected. The classes involved in the *Heartbeat* system are found in the *fr.inria.plasmalab.distributed.algorithm.heartbeat* package.

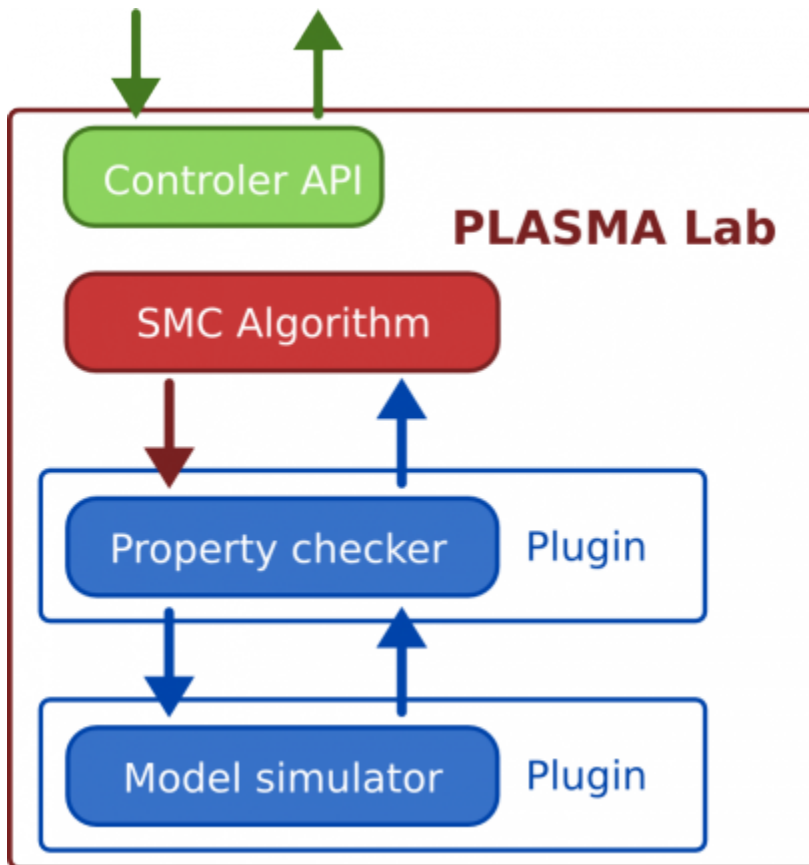
6.6.7 RestLet

Our distributed algorithms use RestLet to handle communication between Scheduler and Workers. RestLet is a Restful framework API for Java.

More details on RestLet API can be found at <http://restlet.com/> or around the Web.

6.7 Controller

The Controller project – *fr.inria.plasmalab.controller* – exposes the PLASMA Lab API. This layer is used to launch and control the execution of a PLASMA Lab experimentation. The PLASMA Lab API is effectively used by the PLASMA Lab GUI.



6.7.1 Initialization

The *Controller* class is a singleton class created by calling the *createController* function. Path to the configuration file and java arguments are passed to the initialization function.

According to the configuration file, the Controller will configure the log4j module and load required PLASMA Lab plugins.

6.7.2 PluginLoader

Loading PLASMA Lab plugins is done by the *PluginLoader* class. Most SMC algorithms are loaded statically by the *PluginLoader* as they are included in the `libs` directory of PLASMA Lab distribution. Other plugins (models, requirements and additional algorithms) are loaded from the plugin sources specified in the configuration.

6.7.3 Experimentation Manager

The *ExperimentationManager* class launches and monitors the execution of an SMC Algorithm. The experimentation manager can be accessed through the Controller.

A new experimentation is initialized by calling the *setupAnExperiment* method. The experimentation is monitored through the *ExperimentationListener* implementation. However the manager only transfers notifications to registered listener. The process calling the Controller API must give a reference to an object implementing the *ExperimentationListener* interface.

6.7.4 Simulation Manager

The *SimulationManager* follows the same purpose as the *ExperimentationManager* but for the simulation mode. As it is only passing calls to simulation functions of the selected model its use may be avoidable.

6.8 Plugins development tutorial

This tutorial explains how to develop new plugins for PLASMA Lab (algorithm, checker or simulator). The sources of this tutorial can be downloaded below:

- [sources.zip](#) are the sources of the new plugins.
- [myalgorithm.jar](#) is the new algorithm plugin produced by the tutorial.
- [mysimulator.jar](#) is the simulator plugin produced by the tutorial.
- [mychecker.jar](#) is the checker plugin produced by the tutorial.
- [testmysim.plasma](#) is a PLASMA project to test the new plugins.

The tutorial projects require the *fr.inria.plasmalab.algorithm* and *fr.inria.plasmalab.workflow* libraries of Plasma Lab, as well as the *jspf.core* library for defining plugins, and the *slf4j-api* library for logging.

The outline of this section is the following:

6.8.1 Algorithm

In this tutorial we will re-implement a simple Monte-Carlo algorithm.

Factory

The factory is used to load the plugin dynamically and specify the parameters of the algorithm. It implements the *InterfaceAlgorithmFactory* and it declares a new JSPF *Plugin* using the annotation `@PluginImplementation` before the class declaration.

```
@PluginImplementation
public class MyAlgorithmFactory implements InterfaceAlgorithmFactory
```

Then we implement the functions of the *InterfaceAlgorithmFactory* interface

- The *id* is a unique identifier for the algorithm. It used to select the algorithm in the CLI and through various components of the GUI.
- The *name* will appear the GUI menus. It should be return by the *toString* method.
- The *description* is a short text displayed in the algorithm selection panel or in the CLI *info* command.

```
@Override
public String getId() {
    return "myalgorithm";
}

@Override
public String getName() {
    return "My Algorithm";
}

@Override
public String getDescription() {
    return "Reimplementation of Monte Carlo.";
}

@Override
public String toString(){
    return getName();
}
```

The parameters of the algorithm are specified by the *getParametersList* method of the factory. It returns a list of *SMCParameter*. A basic *SMCParameter* is defined with a name and a description. It will appear in th GUI as a text field allowing to enter a value. The different types of parameters are presented on the *algorithms* section.

Our algorithm has a single parameter of type *SMCParameter*, with name **Nb sims**, description **Number of simulations**. We set the final value to false to mean that the parameter is not associated to a Boolean value.

```
@Override
public List<SMCParameter> getParametersList() {
    List<SMCParameter> parameters = new ArrayList<SMCParameter>(1);
    parameters.add(new SMCParameter("Nb Sims","Number of simulations",
    ↪ false));
    return parameters;
}
```

The value of the parameters will be given by the GUI or the CLI in a map *Map<String, Object>*. This map is read by the *createScheduler* function when starting a new experiment. The function parses the value of the parameters, checks if they satisfy the requirements of the algorithm and finally instantiates the algorithm.

In our algorithm we parse the value of the “Nb Sims” parameter into an integer and we check if the value is positive.

```
@Override
public InterfaceAlgorithmScheduler createScheduler(AbstractModel model,
    List<AbstractRequirement> reqs,
    Map<String, Object> parametersMap) throws PlasmaParameterException {
    int nbSims = 0;
    try {
        nbSims = Integer.parseInt(parametersMap.get("Nb Sims").
        toString());
    }
    catch(Exception e){
        throw new PlasmaParameterException(e);
    }
    if ( !(nbSims > 0) )
        throw new PlasmaParameterException("Nb Sims" + " must be > 0.
        ");
    return new MyAlgorithm(model, reqs, nbSims, getId());
}
```

Alternatively we can specify the parameters with the *fillParametersMap* method. In that case we parse the value from an array of string, assuming that the values have been given in the right order.

```
@Override
public void fillParametersMap(Map<String, Object> parametersMap,
    String[] parameters) throws PlasmaParameterException {
    try{
        if(parameters.length>=1)
            parametersMap.put("Nb Sims", Integer.
            parseInt(parameters[0]));
        else
            throw new PlasmaParameterException("Not enough
            parameters for MyAlgorithm.");
    } catch(NumberFormatException e){
        throw new PlasmaParameterException(e);
    }
}
```

The other methods of the factory are used to create a distributed algorithm. We will not present this case in this tutorial.

Therefore we return false from the *isDistributed* method, and we return null values from the *createWorker* and *getResourceHandler* methods:

```
@Override
public boolean isDistributed() {
    return false;
}

@Override
public InterfaceAlgorithmWorker createWorker(AbstractModel arg0, List
    <AbstractRequirement> arg1) {
    return null; // not distributed
}
```

(continues on next page)

(continued from previous page)

```

@Override
public Class<?> getResourceHandler() {
    return null; // not distributed
}

```

Algorithm

The main class of our new SMC algorithm extends the *AbstractAlgorithm* class. This class implements some functions from the *InterfaceAlgorithmScheduler* interface that are common between most SMC algorithm.

The *AbstractAlgorithm* class already provides the basic attributes for the algorithm (*model*, *requirements*, *nodeURI*) that we load in the constructor with additionally the number of simulations parameter:

```

public MyAlgorithm(AbstractModel model, List<AbstractRequirement> reqs, int
↳nbSims, String id) {
    this.model = model;
    this.requirements = reqs;
    this.nbSims = nbSims;
    this.nodeURI = id;
}

```

Then we implement the *run* method of the algorithm.

```

@Override
public void run() {
    initializeAlgorithm();
    listener.notifyAlgorithmStarted(nodeURI);
    logger.info("Starting " + nodeURI + " with " + nbSims + " simulations.
↳");

    List<ResultInterface> results = new ArrayList<ResultInterface>(1);
    double result = 0.0;
    try {
        for (int i=1; i<= nbSims && !stopOrderReceived; i++) {
            InterfaceState path = model.newPath();
            double res = requirements.get(0).check(path);
            if (res > 0) {
                result += res;
            }
        }
    }
    catch (PlasmaExperimentException e) {
        //logger.error(e.getMessage(),e);
        listener.notifyAlgorithmError(nodeURI, e.toString());
        errorOccured = true;
    }
    result /= nbSims;
    results.add(new MyResult(requirements.get(0), result, nbSims));

    if(!errorOccured){
        // Notify new results
    }
}

```

(continues on next page)

(continued from previous page)

```

        listener.publishResults(nodeURI, results);
        // Notify completed
        if(stopOrderReceived)
            listener.notifyAlgorithmStopped(nodeURI);
        else
            listener.notifyAlgorithmCompleted(nodeURI);
    }
}

```

- We first call the *initializeAlgorithm* method from *AbstractAlgorithm* in order to perform necessary initializations.
- We can send notifications to the user interface via the *listener* attribute.
- We perform *nbSims* simulations by calling the *model.newPath()* method.
- Each simulation is checked against the (first) requirement using *requirements.get(0).check(path)*
- The algorithm is stopped whenever *stopOrderReceived* is set to true (by the *AbstractAlgorithm* class).
- At the end of the algorithm we create the result (the ratio of the sums of the simulations results with the number of simulations) and we add this to a *ResultInterface* object.
- The list of results (in our case a single one) is send to the user interface via the *listener.publishResults* method.

Result

The final class that we need for our new algorithm is an object to store and send the results of the experimentation.

Our *MyResult* class implements the *SMCResult* interface.

```
public class MyResult implements SMCResult
```

A result is a collection of values associated to an identifier.

In our result we store only two value: the probability and the number of simulations. These two values are the minimal values needed in an *SMCResult*. We create two identifiers associated to these values:

```

private static final ResultIdentifier probaId = new ResultIdentifier(
    ↪ "Probability", false);
private static final ResultIdentifier simId = new ResultIdentifier(
    ↪ "#Simulations", false);

```

We implement a constructor for the result that takes these two values and the requirement that has been checked.

```

public MyResult(AbstractRequirement req, double proba, int nbsims) {
    this.origin = req;
    this.probability = proba;
    this.nbsimulations = nbsims;
}

```

Then we implement the methods of the *SMCResult* interface.

The *getCategory* method returns the name of the requirement that has been checked:

```
@Override
public String getCategory() {
    return origin.getName();
}
```

The *getHeaders* method returns an array of the identifiers of the result:

```
@Override
public InterfaceIdentifier[] getHeaders() {
    InterfaceIdentifier[] ret = new InterfaceIdentifier[2];
    ret[0] = probaId;
    ret[1] = simId;
    return ret;
}
```

Two *getValueOf* methods allow to get the values associated to an identifier, either using the name of the identifier or directly the identifier object:

```
@Override
public Object getValueOf(String header) throws PlasmaExperimentException {
    if (header == probaId.getName())
        return probability;
    else if (header == simId.getName())
        return nbsimulations;
    else
        throw new PlasmaExperimentException("header " + header + " not
↳found in MyResult.");
}

@Override
public Object getValueOf(InterfaceIdentifier id) throws
↳PlasmaExperimentException {
    if (id == probaId)
        return probability;
    else if (id == simId)
        return nbsimulations;
    else
        throw new PlasmaExperimentException("header ID " + id.
↳getName() + " not found in MyResult.");
}
```

The last getters return the requirement, the probability and the number of simulations:

```
@Override
public AbstractRequirement getOriginRequirement() {
    return origin;
}

@Override
public double getPr() {
    return probability;
}
```

(continues on next page)

(continued from previous page)

```
@Override
public int getTotalCount() {
    return nbsimulations;
}
```

6.8.2 Data factory

The first thing to add a new simulator or a new checker is to implement a *factory* interface. The factory interfaces are as follows:

```
public interface AbstractDataFactory extends Plugin
public abstract class AbstractModelFactory implements AbstractDataFactory
public abstract class AbstractRequirementFactory implements AbstractDataFactory
```

- *AbstractDataFactory* creates a data with no semantics.
- *AbstractModelFactory* creates a new simulator.
- *AbstractRequirementFactory* creates a new checker.

Each Factory implementation declares a new JSPF *Plugin*. It is tag with `@PluginImplementation` before the class declaration.

We implement a new factory with one of the following declarations:

```
@PluginImplementation
public class MySimulatorFactory extends AbstractModelFactory
```

```
@PluginImplementation
public class MyCheckerFactory extends AbstractRequirementFactory
```

Then we implement the functions of the *AbstractDataFactory* interface:

- The *name* of the factory appears in the GUI menus
- The *description* is a short text displayed by the GUI *About windows* and the CLI *info* command.
- The *id* is a unique identifier used to identified the data type and the factory in the CLI, in the saved projects and in the distributed services.

```
@Override
public String getName() {
    return "My plugin";
}

@Override
public String getDescription() {
    return "This is a tutorial plugin";
}

@Override
public String getId() {
    return "myplugin";
}
```

The main purpose of the Factory interface is to provide a way of instantiating a simulator or a checker without knowing their classes. This is done by implementing the next two methods (inherited from their respective super class).

```
@Override
public AbstractModel createAbstractModel(String name) {
    return new MySimulator(name, "", getId());
}

@Override
public AbstractModel createAbstractModel(String name, File file) throws
↳ PlasmaDataException {
    return new MySimulator(name, file, id);
}

@Override
public AbstractModel createAbstractModel(String name, String content) {
    return new MySimulator(name, content, getId());
}
```

```
@Override
public AbstractRequirement createAbstractRequirement(String name) {
    return new MyChecker(name, "", getId());
}

@Override
public AbstractRequirement createAbstractRequirement(String name, File file)
↳ throws PlasmaDataException {
    return new MyChecker(name, file, getId());
}

@Override
public AbstractRequirement createAbstractRequirement(String name, String
↳ content) {
    return new MyChecker(name, content, getId());
}
```

And that's all for the factory!

6.8.3 Simulator

In PLASMA Lab the concepts of simulator and model are mixed together. We could say that a model executes itself. For this reason, a simulator inherits from the *AbstractModel* class, that in turn inherits from the *AbstractData* class. The *AbstractData* class describes an object, model or requirement, editable in the PLASMA Lab GUI edition panel. The *AbstractModel* class adds simulation methods.

In this section, we explain some of the implementation needed for a new simulator plugin. The language executed by our tutorial simulator is a succession of + and -. Starting from 0 it will add or remove one to the single value of the state. For instance the model +++-- will produce a trace 0 1 2 3 2 1. Of course this language has no stochastic property.

We only cover a few important methods.

Configuration

The *checkForErrors* function is called before each experimentation/simulation and when modifying the content of the edition panel in the GUI. The purpose of this function is double: to detect any syntax error and to build the model before running it.

Note: the checkForErrors function is part of AbstractData.

Our *checkForErrors* function checks if the sequence contains any other characters than “+”, “-”. In the eventuality of a syntax error, a *PlasmaSyntaxException* is added to the list of errors.

```
@Override
public boolean checkForErrors() {
    // Empty from previous errors
    errors.clear();

    // Verify model content
    InputStream is = new ByteArrayInputStream(content.getBytes());
    br = new BufferedReader(new InputStreamReader(is));
    try {
        while(br.ready()){
            int c = br.read();
            if(!(c=='+'||c=='-')){
                errors.add(new PlasmaSyntaxException(Character.
→toString((char)c)+" is not a valid command"));
            }
        }
    } catch (IOException e) {
        errors.add(new PlasmaDataException("Cannot read model content",
→e));
    }
    initialState = new MyState(0,0);
    return !errors.isEmpty();
}
```

We also create the initial state which will be used to initialize each trace. We could specify a different initial value for an experiment, for instance using optimization declaration in BLTL. This would call the *setValueOf* method to modify the initial state.

```
@Override
public void setValueOf(Map<InterfaceIdentifier, Double> update) throws
→PlasmaSimulatorException {
    // This method change the initial state with a set of initial values.
    for(InterfaceIdentifier id:update.keySet())
        initialState.setValueOf(id, update.get(id));
}
```

New Path

The *content* String is inherited from *AbstractData* and contains the text entered in the edition panel. In our simulator we initialize a stream reader to read *content* character by character. We also initialize the trace with the initial state created by the *checkForErrors* methods.

```
@Override
public InterfaceState newPath() {
    trace = new ArrayList<InterfaceState>();
    trace.add(initialState);
    InputStream is = new ByteArrayInputStream(content.getBytes());
    br = new BufferedReader(new InputStreamReader(is));
    return initialState;
}
```

Simulate

In the simulate method we read the next character. In our language, “+” (resp. “-”) **add** 1 to the value (resp. **subtract** 1 to the value). We also add 1 to the time. We build a new state with the new values. If there is no more character to read we throw a *PlasmaDeadlockException* instead of adding a new state to the trace.

```
@Override
public InterfaceState simulate() throws PlasmaSimulatorException {
    try {
        if (!br.ready())
            throw new PlasmaDeadlockException(getCurrentState(),
        ↪getTraceLength());
        else {
            int c = br.read();
            InterfaceState current = getCurrentState();
            double currentV = current.getValueOf(VALUEID);
            double currentT = current.getValueOf(TIMEID);
            if(c=='+')
                trace.add(new MyState(currentV+1,currentT+1));
            else if(c=='-')
                trace.add(new MyState(currentV-1,currentT+1));
        }
    } catch (IOException e) {
        throw new PlasmaSimulatorException(e);
    }
    return getCurrentState();
}
```


State

A state object is used to store the values of the model. It inherits from the *InterfaceState* interface. Our state object is pretty simple as we store only two variables, the time and the value.

```
public class MyState implements InterfaceState {

    double value;
    double time;

    public MyState(double value, double time) {
        this.value = value;
        this.time = time;
    }
}
```

The *InterfaceState* declares several ways of accessing and setting values. *InterfaceIdentifier* objects are the main way to identify objects (e.g. variable, constant) through the components of PLASMA Lab. The *InterfaceState* interface also provides a method for accessing a value by the string of its identifier.

```
@Override
public Double getValueOf(InterfaceIdentifier id) throws
↳PlasmaSimulatorException {
    if(id.equals(MySimulator.VALUEID))
        return value;
    else if(id.equals(MySimulator.TIMEID))
        return time;
    else
        throw new PlasmaSimulatorException("Unknown identifier: "+id.
↳getName());
}

@Override
public Double getValueOf(String id) throws PlasmaSimulatorException {
    if(id.equals(MySimulator.VALUEID.getName()))
        return value;
    else if(id.equals(MySimulator.TIMEID.getName()))
        return time;
    else
        throw new PlasmaSimulatorException("Unknown identifier: "+id);
}

@Override
public void setValueOf(InterfaceIdentifier id, double value) throws
↳PlasmaSimulatorException {
    if(id.equals(MySimulator.VALUEID))
        this.value = value;
    else if(id.equals(MySimulator.TIMEID))
        this.time = value;
    else
        throw new PlasmaSimulatorException("Unknown identifier: "+id.
↳getName());
}
```

Except getters and setters, the relation between state objects and their associated model is free. As their can be a large

number of state for a single model instance, we recommend to keep the memory usage of states as low as possible.

Identifier

Identifiers are a shared objects to identify values (e.g. variable, constant) through different components of PLASMA Lab. As our model has only two variables, we declare them as static identifiers in the simulator.

```
protected static final MyId VALUEID = new MyId("X"); //VALUE
protected static final MyId TIMEID = new MyId("#"); //TIME
```

An external component, such as a checker component, access the identifier of the model through *getIdentifiers*. This creates a map to store the identifiers and sort them according to their name.

```
@Override
public Map<String, InterfaceIdentifier> getIdentifiers() {
    Map<String, InterfaceIdentifier> map = new HashMap<String,
↪InterfaceIdentifier>();
    map.put(TIMEID.getName(), TIMEID);
    map.put(VALUEID.getName(), VALUEID);
}
```

We could then write a BLTL property checking if the value of the model reached a given threshold after 10 simulation steps: $F \leq \#10 \ X > 5$. This uses *getTraceLength* function to measure the number of steps.

Our model also contains its own notion of time implemented by **TIMEID**. In the simulator we implement the methods *getTimeId* and *hasTime* to notify that the model has its own time.

```
@Override
public InterfaceIdentifier getTimeId() {
    return TIMEID;
}

@Override
public boolean hasTime() {
    //Return true if model as a dedicated time value.
    //Otherwise only trace length is used.
    return true;
}
```

Then, we could check the property using the time implemented in the states instead of the steps: $F \leq 10 \ X > 5$. This would return the same results as the previous property since time increases at the same pace than steps (we add 1 to time at each step).

6.8.4 Checker

In PLASMA Lab the concepts of checker and requirement are mixed together. We could say that a requirement checks itself. For this reason, a checker inherits from the *AbstractRequirement* class, that in turns inherits from the *AbstractData* class. The *AbstractRequirement* class adds verification methods to the *AbstractData* class.

In this section, we explain some of the implementation needed for a new checker plugin. The language verified by our plugin contain two values. The first is a time stamp *TIME* and the second a value *VAL*. the checker checks a trace of the model when it reaches time *TIME*. If the value of the model variable “X” is strictly superior to *VAL*, the requirement is evaluated to true. Otherwise the evaluation returns false.

Configuration

The *checkForErrors* function is called before each experimentation/simulation and when modifying the content of the edition panel in the GUI. The purpose of this function is double: to detect any syntax error and to build the requirement before running it.

Our *checkForErrors* function checks if the sequence contains at least two integer values: one for the time bound, the other for the threshold. A *PlasmaSyntaxException* is added to the list of errors if an error is found.

```
@Override
public boolean checkForErrors() {
    errors.clear();
    String[] splitted = content.split(" ");
    try{
        if(splitted.length < 2)
            errors.add(new PlasmaSyntaxException("The checker needs a time and
→a threshold value"));
        until = Integer.valueOf(splitted[0]);
        threshold = Integer.valueOf(splitted[1]);
    } catch(Exception e){
        errors.add(new PlasmaDataException("Exception occurred while parsing
→requirement", e));
    }
    return !errors.isEmpty();
}
```

We also associate the requirement to model with the *setModel* method. This method is called before *checkForErrors* when starting an experimentation/simulation. It will allow usually to get the identifiers of the model and then check if the variables use in the requirement correspond to variables of the model.

With the *setModel* method we get the model and the *TIMEID* used by the model to count time.

```
@Override
public void setModel(AbstractModel model) {
    this.model = model;
    this.TIMEID = model.getTimeId();
}
```

Note: before launching an experimentation/simulation no model is associated to the requirement. Then, the requirement *checkForErrors* function may not be able to find the variable of the model and can produce errors.

Check

The *check* method computes the results of checking the requirement on a trace. This results is represented by a double value. If the result should only be a Boolean then the method should return 0 if false, and 1 if true.

Our requirements check if the value “X” is superior to a threshold after *TIME* timestamp was reached. We evaluate To evaluate this property, we simulate the trace up to the needed time by calling the *simulate* method. To measure time we use the *TIMEID* provided by the model. Once enough states have been generated, we check the variable against the threshold and return the result.

```
@Override
public Double check(InterfaceState path) throws PlasmaCheckerException {
```

(continues on next page)

(continued from previous page)

```

    try {
        while(model.getCurrentState().getValueOf(TIMEID)<until)
            model.simulate();
        return (model.getCurrentState().getValueOf("X") > threshold ? 1.0 : 0.0);
    }
    catch (PlasmaSimulatorException e) {
        throw new PlasmaCheckerException(e);
    }
}

```

There are two other check functions. The first is used in simulation mode. In this mode the user has a direct control over the simulation to add or remove states from the trace. The check function specifies an *untilStep* parameter that defines the length of the trace that must be checked. If the requirement cannot check the requirement within this length and more state are needed, the checker returns a NaN value meaning “undecided”.

Note: Here *untilStep* refers to the step time, not the time implemented in the model.

Therefore our checker with *untilStep* parameter only checks the model current path and does not simulate new state.

```

@Override
public Double check(int untilStep, InterfaceState path) throws PlasmaCheckerException {
    try {
        for(int step=0; step < untilStep || step >= model.getTraceLength(); step++){
            InterfaceState state = model.getStateAtPos(step);
            if(state.getValueOf(TIMEID) == until)
                //Time bound reached
                return (model.getCurrentState().getValueOf("X") > threshold ? 1.0 : 0.0);
        }
    }
    catch (PlasmaSimulatorException e) {
        throw new PlasmaCheckerException(e);
    }
    //Untilstep reached or not enough state to reach time bound
    return Double.NaN;
}

```

The final check function takes has parameters an identifier and a value. It checks the trace until it is decided or the identifier reaches the given value. This function is used by the importance splitting algorithm only, and may not be implemented otherwise.

DEVELOPMENT

7.1 PLASMA Lab @ INRIA

The PLASMA Lab project is developed and hosted by INRIA. In this section we describe the various support tools provided by INRIA.

7.1.1 GForge

PLASMA Lab is hosted on the [INRIA Forge](#). The forge provides a git repository, an HTTP server and access to bug tracking tools and mailing list.

To access these services an INRIA Forge account is required. The INRIA Forge does not use the LDAP account.

Git

Git is our Version Control System. The Git documentation can be accessed on:

- <http://git-scm.com/book/fr>

Only developers registered on the forge can access to the Git repository. SSH must be installed on your computer.

Tracking tools

- [Bugs and issues tracking tool](#)
- [Technical solutions tracking tool](#)

Support

- [PLASMA Lab support mailing list](#)

GForge hosting

The INRIA GForge provides hosting capacity. The PLASMA Lab GForge website is located at <http://plasma-lab.gforge.inria.fr/>. We use this to host the Javadoc and this manual but also as a repository for PLASMA Lab bundles and examples library.

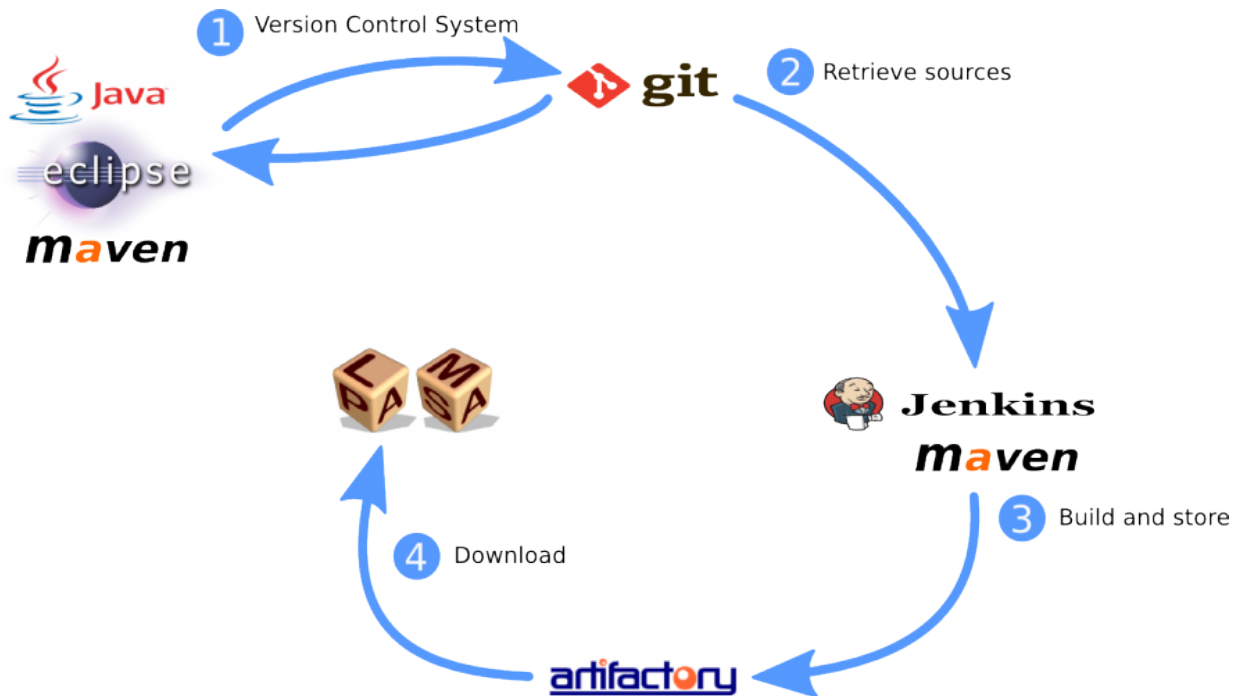
7.1.2 Web site

The main PLASMA Lab website is hosted on <https://project.inria.fr/plasma-lab/>. It is a wordpress blog/site and can be modified by accessing to the [administration interface](#).

project.inria.fr depends from a different service than GForge.

7.2 Continuous integration

The PLASMA Lab project uses Continuous Integration practice. After an overview on this practice and the tool chain we use, this chapter details the procedure for publishing a new version of PLASMA Lab.



7.2.1 Git

Git is our Version Control System. The Git repository is hosted on gforge.inria.fr and a link to the PLASMA Lab Git can be retrieved by joining the PLASMA Lab Gforge project.

- <http://git-scm.com/>
- <http://gforge.inria.fr/>

7.2.2 Maven

Maven handles project dependencies and build configuration.

- <http://maven.apache.org/>

7.2.3 Jenkins

Jenkins is an Automated Continuous Integration server. The server is hosted at `ci.inria.fr`. Jenkins is used to launch automated SNAPSHOT build every night as well as manual release build. Executables are the deployed to Artifactory.

- <http://jenkins-ci.org/>
- <https://ci.inria.fr/plasmalab/>
- <https://wiki.inria.fr/ciportal/>

From version 1.4.4 Jenkins is not used in the Continuous Integration process due to uncompatibilities between Jenkins and Maven Artifactory.

7.2.4 Artifactory

Artifactory is an open-source repository manager. Build are stored on it and can be downloaded by PLASMA Lab users. Artifactory is also hosted by INRIA.

- <http://maven.inria.fr/artifactory>

7.2.5 Release Procedure (from version 1.4.4)

The procedure is run from the directory `fr.inria.plasmalab.root`. All links in the procedure are relative to this directory.

1 Prepare the release

Run the script `prepare.sh` with the new release version and the next snapshot version:

- The script updates the `.bat` scripts and the `README.md` with the new release version and it commit these changes.
- It then runs the maven release plugin (`mvn release:prepare`) to update the `pom.xml`:
 - It updates the `pom.xml` files to the new release version.
 - It creates a `git-tag` whose name is the release version.
 - Then it updates again the `pom.xml` files to the new development version.
 - It commits the changes.
- Afterwards, the script updates again the `.bat` scripts and the `README.md` with the snapshot version.
- It commits the changes and pushes to the git repository.

2 Build and deploy

Retrieve the tagged release with `git checkout release-version` and then run the script `deploy.sh`:

- This runs maven with the assembly, javadoc and deploy plugins (`mvn clean package assembly:assembly javadoc:javadoc deploy:deploy`):
 - It compiles the `.jar` artifacts for all the projects (listed in the `pom.xml` file).
 - It compiles distribution bundles with the assembly plugin (using the configuration in `src/assembly/assembly.xml`) in `target`.
 - It compiles the javadoc in `target/site`.
 - It deploys the artifacts to `maven.inria.fr/artifactory/plasmalab-public-release`.
- It deploys the distribution bundles to Gforge.
- It deploys the javadoc to Gforge.

Retrieve the head of the repository `git checkout master` and re-run the script `deploy.sh` to deploy the snapshot version.

3 Update the documentation

- Update the content related to the new version.
- Check links in the documentation that may refer to the old version.
- Update PLASMA Lab's version number in `conf.py`.
- Compile with `make html` and `make latexpdf`.
- Update the pages `index.php` and `manuel.php` with the new version.
- Deploy: run the script `deploy.py` to show which commands to launch. The commands will create a new directory on the gforge online repository and upload in it the documentation (the builds).

4 Update “external” content

This concerns content that is not automatically updated:

- Update Plasma2Simulink.
- Update the tutorials if needed.
- Update the PTA plugin if needed.

5 Update the website

- Update the version number and the content of the website (if needed). The documentation links are now relative to php files and don't need to refer to the version.
- Update the download links.
- Post a news.
- Send a mail to the mailing lists (`plasma-lab-news@lists.gforge.inria.fr` and `plasma-lab-developers@lists.gforge.inria.fr`).

7.2.6 Snapshot Release Procedure (from version 1.4.4)

Run the `deploy.sh` script from `fr.inria.plasmalab.root`.