



**HAL**  
open science

# Plasma Lab Statistical Model Checker: Architecture, Usage and Extension

Axel Legay, Louis-Marie Traonouez

► **To cite this version:**

Axel Legay, Louis-Marie Traonouez. Plasma Lab Statistical Model Checker: Architecture, Usage and Extension. SOFSEM 2017 - 43rd International Conference on Current Trends in Theory and Practice of Computer Science, Jan 2017, Limerick, Ireland. hal-01613581

**HAL Id: hal-01613581**

**<https://hal.science/hal-01613581>**

Submitted on 9 Oct 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Plasma Lab Statistical Model Checker: Architecture, Usage and Extension

Axel Legay and Louis-Marie Traonouez

Inria, Rennes, France

**Abstract.** Plasma Lab is a modular statistical model checking (SMC) platform that facilitates multiple SMC algorithms and multiple modelling and query languages. Plasma Lab may be used as a stand-alone tool with a graphical development environment or invoked from the command line for high performance scripting applications. This tutorial first presents an overview of Plasma Lab architecture, modelling languages and algorithms. Then we present the usage of the tool to design models and perform various experiments Finally we propose a tutorial on how to develop a new plugin for Plasma Lab.

## 1 Introduction

Statistical model checking (SMC) employs Monte Carlo methods to avoid the state explosion problem of probabilistic (numerical) model checking. To estimate probabilities or rewards, SMC typically uses a number of statistically independent stochastic simulation traces of a discrete event model. Being independent, the traces may be generated on different machines, so SMC can efficiently exploit parallel computation. Reachable states are generated on the fly and SMC tends to scale polynomially with respect to system description. Properties may be specified in bounded versions of the same temporal logics used in probabilistic model checking. Since SMC is applied to finite traces, it is also possible to use logics and functions that would be intractable or undecidable for numerical techniques. In recent times, dedicated SMC tools, such as YMER<sup>1</sup>, VESPA, APMC<sup>2</sup> and COSMOS<sup>3</sup>, have been joined by statistical extensions of established tools such as PRISM<sup>4</sup>, UPPAAL<sup>5</sup> and MRMC<sup>6</sup>. In this work we describe Plasma Lab<sup>7</sup>, a modular Platform for Learning and Advanced Statistical Model checking Algorithms [2].

SMC approximates the probabilistic model checking problem by estimating the parameter of a Bernoulli random variable, for which there are well defined confidence bounds (e.g., [10]). The general principle is to simulate the model or system in order to generate execution traces. These traces are checked with respect to a logic such as Bounded Linear Temporal Logic (BLTL) [1] and the results are combined with statistical techniques.

<sup>1</sup> [www.tempastic.org/ymer/](http://www.tempastic.org/ymer/)

<sup>2</sup> <http://archive.is/OKwMY>

<sup>3</sup> [www.lsv.ens-cachan.fr/~barbot/cosmos/](http://www.lsv.ens-cachan.fr/~barbot/cosmos/)

<sup>4</sup> [www.prismmodelchecker.org](http://www.prismmodelchecker.org)

<sup>5</sup> [www.uppaal.org](http://www.uppaal.org) <sup>6</sup> [www.mrmc-tool.org](http://www.mrmc-tool.org) <sup>7</sup> <https://project.inria.fr/plasma-lab/>

BLTL restricts the classical Linear Temporal Logic by bounding the scope of the temporal operators. Syntactically, we have

$$\varphi, \varphi' := \text{true} \mid P \mid \varphi \wedge \varphi' \mid \neg\varphi \mid X_{\leq t} \mid \varphi U_{\leq t} \varphi',$$

where  $\varphi, \varphi'$  are BLTL formulas,  $t \in \mathbb{Q}_{\geq 0}$ , and  $P$  is an atomic proposition that is valid in some state. As usual, we define  $F_{\leq t}\varphi \equiv \text{true} U_{\leq t}\varphi$  and  $G_{\leq t}\varphi \equiv \neg F_{\leq t}\neg\varphi$ . The semantics of BLTL, presented in Table 1, is defined with respect to an execution trace  $\omega = (s_0, t_0), (s_1, t_1), \dots, (s_n, t_n)$  of the system, where each state  $(s_i, t_i)$  comprises a discrete state  $s_i$  and a time  $t_i \in \mathbb{R}_{\geq 0}$ . We denote by  $\omega^i = (s_i, t_i), \dots, (s_n, t_n)$  the suffix of  $\omega$  starting at step  $i$ .

$\omega \models X_{\leq t} \varphi$ iff $\exists i, i = \max\{j \mid t_0 \leq t_j \leq t_0 + t\}$ and $\omega^i \models \varphi$	
$\omega \models \varphi_1 U_{\leq t} \varphi_2$ iff $\exists i, t_0 \leq t_i \leq t_0 + t$ and $\omega^i \models \varphi_2$ and $\forall j, 0 \leq j < i, \omega^j \models \varphi_1$	
$\omega \models \varphi_1 \wedge \varphi_2$ iff $\omega \models \varphi_1$ and $\omega \models \varphi_2$	$\omega \models \neg\varphi$ iff $\omega \not\models \varphi$
$\omega \models P$ iff $s_i \models P$	$\omega \models \text{true}$

Table 1: Semantics of BLTL.

Plasma Lab implements qualitative and quantitative SMC algorithms. Qualitative algorithms decide between two contrary hypothesis (e.g. is the probability to satisfy the requirement is above a given threshold), while quantitative techniques compute an estimation of a stochastic measure (e.g., the probability to satisfy a property).

The “crude” Monte Carlo algorithm is a quantitative technique that uses  $N$  simulation traces  $\omega_i, i \in \{1, \dots, N\}$ , to calculate  $\tilde{\gamma} = \sum_{i=1}^N \mathbf{1}(\omega_i \models \varphi)/N$ , an estimate of the probability  $\gamma$  that the system satisfies a logical formula  $\varphi$ , where  $\mathbf{1}(\cdot)$  is an indicator function that returns 1 if its argument is **true** and 0 otherwise. Using the Chernoff-Hoeffding bound [10], setting  $N = \lceil (\ln 2 - \ln \delta)/(2\epsilon^2) \rceil$  guarantees the probability of error is  $\Pr(|\tilde{\gamma} - \gamma| \geq \epsilon) \leq \delta$ , where  $\epsilon$  and  $\delta$  are the precision and the confidence, respectively.

## 2 Plasma Lab 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 tool architecture is displayed in Fig. 1. The core of Plasma Lab is a light-weight controller that manages the experiments and the distribution mechanism.

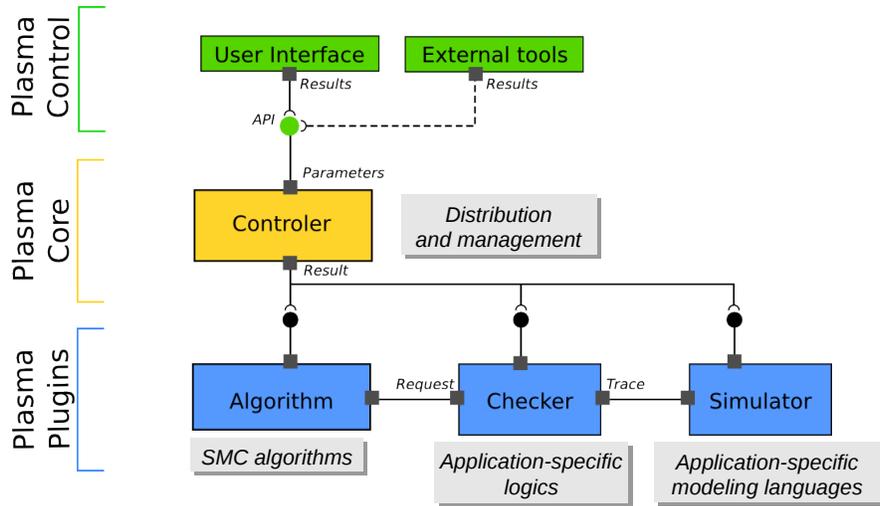


Fig. 1: Plasma Lab architecture.

It implements an API that allows to control the experiments either through user interfaces or through 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. Only a few classes must be implemented to extend the tool with custom plugins for adding new languages or checkers.

An SMC algorithm collects samples obtained from a checker component. The checker asks the simulator to initialize a new trace. Then, it controls the simulation by requesting new states, with a *state on demand* approach: new states are generated only when needed to decide the property. Depending on the property language, the checker either returns Boolean or numerical values. Finally, the **algorithm** notifies the progress and sends the results through the **controller** API. Plasma Lab offers several advanced SMC algorithms that can be applied to various models, including SMC algorithms for rare events and nondeterministic models.

Tables 2,3,4 presents the list of simulator, checker and algorithms plugins currently available with Plasma Lab. Plasma Lab has also been used to verify other types of models through a connection or an integration with other tools.

### 3 Tool Usage

Plasma Lab also includes several user interfaces capable of launching SMC experiments through the **controller** API, either as standalone applications or integrated with external tools:

RML	Reactive Module Language: input language of the tool Prism for Markov chains models
RML Adaptive	Extension of RML for adaptive systems
Bio	Biological language for writing chemical reactions
Matlab Session	Allows to control the simulator of Matlab/Simulink
SystemC	Simulation of SystemC models. The plugin requires an external tool (MAG, <a href="https://project.inria.fr/pscv/">https://project.inria.fr/pscv/</a> ) to instrument SystemC models and generate a C++ executable used by the plugin.

Table 2: Plasma Lab simulators

BLTL	Bounded Linear Temporal Logic
ALTL	Adaptive Linear Temporal Logic, and extension of BLTL with new operators for adaptive systems
GSCL	Goal and Contract Specification Language, a high level specification language for systems of systems
Nested	BLTL checker enhanced with nested probability operator
RML Observer	A plugin that allows to write requirement as observers using a language similar to RML. It is used to write rare properties.

Table 3: Plasma Lab checkers

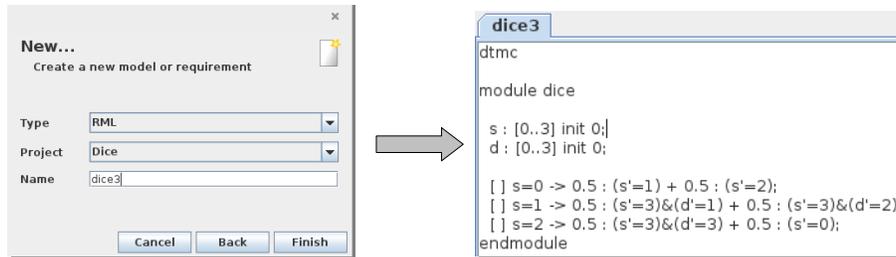
Monte Carlo	Monte Carlo probability estimation with Chernoff-Hoeffding bound [10].
SPRT	Sequential Probability Ratio Test for hypothesis testing [11].
Importance splitting	Estimate the probability of rare events using the importance splitting technique [5,6,7] to decompose a requirement with low probability into a product of higher conditional probabilities that are easier to estimate.
Importance sampling with cross entropy	Estimate the probability of rare events using importance sampling [4], to weight the probability distribution of the original system to favour the rare event, and cross entropy, to determine an optimal weighted distribution.
SMC for nondeterministic models	“Smart sampling” algorithms [3,9] to estimate minimum and maximum probabilities in non deterministic models.

Table 4: Plasma Lab algorithms

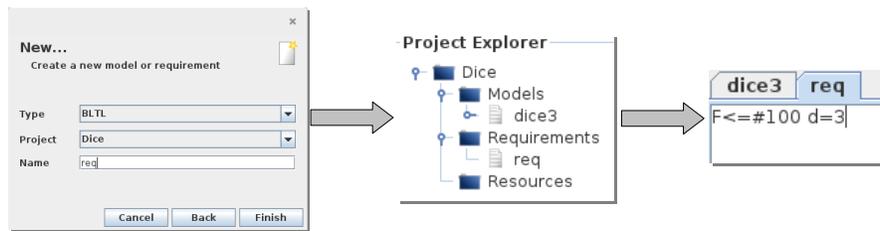
- Plasma Lab Graphical User Interface (GUI). This is the main interface of Plasma Lab. It incorporates all the functionalities of Plasma Lab and allows to open and edit PLASMA project files.
- Plasma Lab Command Line (CLI). A terminal interface for Plasma Lab, with experiment and simulation functionalities, that allows to incorporate Plasma Lab algorithms into high performance scripting applications.
- Plasma Lab Service. A graphical or terminal interface for Plasma Lab distributed service. Its purpose is to be deployed on a remote computer to run distributed experiments, in connection with the Plasma Lab main interface.
- PLASMA2Simulink. This is a small “App” running from Matlab that allows to launch Plasma Lab SMC algorithms directly from Simulink.

We present the usage of the GUI to design a RML model, simulate it and verify a BLTL requirement. We also present the commands of the CLI that perform the same experiments. The GUI is composed of several panels that allow (i) to load, create and edit projects that comprise models and requirements, (ii) to perform simulations and debugging step-by-step, and (iii) to perform various forms of SMC experimentation and optimization, either locally or using distributed algorithms.

### 3.1 Modelling

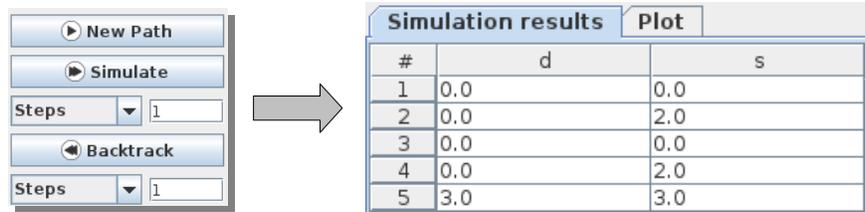


We begin a new project and we add a new model of type RML to the project. In the edition panel we write the content of the model. The model that we design is a discrete time Markov chain (dtmc) of a 3-sided dice, implemented with fair-coins. RML models consist in a set of modules. Each module is a Markov chain that consists in a set of variables and a set of transitions defined with guarded commands. Modules can interact through global variables or transition synchronizations. Our model consists in a single module, with two integers  $s$  and  $d$ , that define a state and the value of the dice, respectively. We use three guarded commands to implement the fair-coin tossing. Each command consists in guard that is a Boolean expression (e.g.  $s = 0$ ), and a set of actions that define the next state (e.g.  $(s' = 3) \& (d' = 1)$ ) and a probability (always 0.5 in our model, as we only use fair-coins).



We now add a BLTL requirement to the project that appears in the project explorer panel. In the edition panel we write the BLTL property. With this requirement we want to determine the probability of drawing the value 3 within 100 steps.

### 3.2 Simulation



The simulation panel allows to execute a model step by step while observing the evolution of the variables. It is useful in the design process of a model for testing and debugging. The simulation panel consists in a control panel that allows to start a new simulation (New Path), add one or more states to the trace (Simulate), or remove states (Backtrack). The simulation results display the current trace with the step number (#) and the values of the variables (s and d).

We can also use the CLI to simulate the model with the following command `./plasmacli.sh simu -m dice.rml:rml` This starts an interactive console. We ask for 10 steps, it produces the following trace, that has reached a deadlock after 2 steps:

---

```
Simulation started.
Enter step number (default is 1), r to restart and q to quit.
#  d  s
10
1  0.0 2.0
2  3.0 3.0
Deadlock reached at state #3:
s: 3.0;
d: 3.0;
```

---

### 3.3 Experimentation

The interface shows a 'File selection' panel with 'Project: Dice', 'Model: dice3', and 'Requirements: req'. Below it is an 'Algorithm:' panel with 'Algorithm: Chernoff', 'Epsilon: 0.01', and 'Delta: 0.01'. An arrow points to the 'Experimentation results' table.

#	Name	# Simulations	# Positive Simulation	Result
1	req	26492	8812	0.3326287181...

The experimentation panel allows to run experiments using using an SMC algorithm. We select a model and a requirement, then in the algorithm panel we select the algorithm and we configure its parameters. We select for instance the Monte Carlo algorithm with the Chernoff bound. We configure the parameters Epsilon (precision) and Delta (confidence). The results show an estimate of the probability to satisfy the BLTL property on our model.

We can also use the following command to launch this experiment:

```
./plasmacli.sh launch -m dice.rml:rml -r dice.bltl:bltl
-a chernoff -A Epsilon=0.01 -A Delta=0.01
```

---

Name	# Simulations	# Positive Simulation	Result
dice.bltl	26492	8762	0.3307413558810

We now add to our model a reward to count the number of coin flips before reaching a final decision. We add a "rewards" structure in our RML model and we add a new BLTL requirement that computes the expected number of coin flips when the BLTL property is satisfied:

```
rewards "coin_flips"
    [] s<7 : 1;
endrewards
```

---

```
R { "coin_flips" } F<=#100 s=3
```

The result of the Monte Carlo experiment is now the expected value of the reward, instead of a probability:

#	Name	# Simulations	# Positive Simulation	Result
1	2.64flips	26492	26492	2.654839196738638

### 3.4 Distributed SMC Experiments

Plasma Lab API provides generic methods to define distributed algorithms, which are a significant advantage of the SMC approach. The distribution of the experiments is implemented with Restlet technology, using the architecture presented in Fig. 2. The main interface of Plasma Lab launches an SMC algorithm scheduler, while a series of services are launched on remote computers.

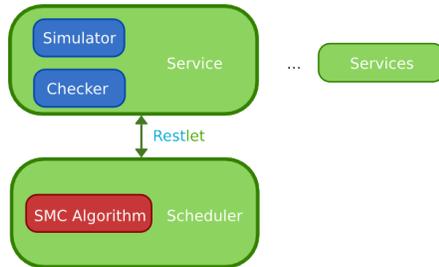


Fig. 2: Distributed architecture.

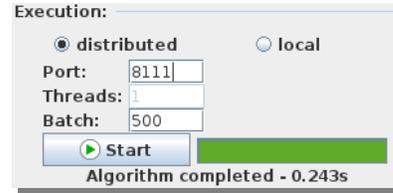


Fig. 3: Execution panel.

Each service is loaded with a copy of the model simulator and a copy of the property checker. Then, the scheduler sends work orders to the services, via Restlet. These orders consist in performing a certain number of simulations and checking them with the checker. When a service has finished its work it sends the result back to the scheduler. According to the SMC algorithm, the scheduler either displays the results via the interface or decides that more work is needed.

We can configure the distributed mode in the GUI in the execution panel presented in Fig. 3. . In local mode the algorithm runs the simulations locally on one or several threads. In distributed mode the algorithm starts a scheduler and waits for clients to connect. It will then request the clients to perform a number of simulations configured with the Batch parameter and then it waits for the results.

### 3.5 SMC and Nondeterminism

Classical SMC algorithms can only be apply to fully stochastic models, such as Markov chains. With RML we can design Markov decision processes (MDP) that interleave probabilistic transitions with nondeterministic transitions. The choice and the order of nondeterministic transitions can radically affect the probability to satisfy a given property or the expected reward. Since it is useful to evaluate the upper and lower bounds of these quantities, we are interested in finding the optimal *schedulers* that do this.

We reuse our previous model of a 3-valued dice and we construct a MDP with two dices. We will roll each dice once and we allow to re-roll one dice. We would like to estimate the minimum and maximum probabilities that both dices return the same value. We modify our RML model accordingly:

---

```
mdp

global r : [0..1] init 0;

module dice
  s : [0..3] init 0;
  d : [0..3] init 0;

  [ ] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
  [ ] s=1 -> 0.5 : (s'=3)&(d'=1) + 0.5 : (s'=3)&(d'=2);
  [ ] s=2 -> 0.5 : (s'=3)&(d'=3) + 0.5 : (s'=0);
```

```

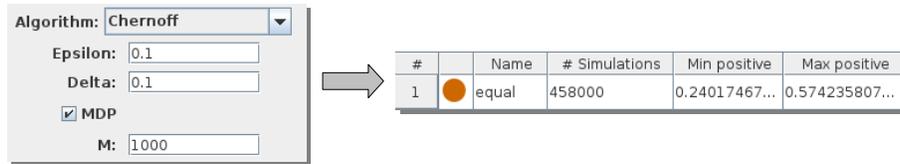
[ ] s=3 & r=0 -> (s'=0)&(r'=1);
[ ] s=3 & r=0 -> (r'=1);
endmodule

module dice2 = dice[s=s2,d=d2] endmodule

```

We change the type from `dtmc` to `mdp`. Then we add a global variable `r` that will count the number of re-roll (maximum 1). In the dice module, we add two commands, one that allows to re-roll the dice, and one that decide to stop. Since these two commands are enabled at the same condition (`s=3 & r=0`) the choice between the two is nondeterministic. Finally we add a second dice that is a copy of the first module in which the variables `s` and `d` are renamed `s2` and `d2`. The order in which the two dices will be rolled is also nondeterministic.

Several algorithms of Plasma Lab can be use with nondeterministic models. When using Monte Carlo algorithms, the default behavior is to simulate the model using a uniform distribution for nondeterministic choices. However the MDP option in the Chernoff algorithm allows to specify the number of schedulers to evaluate. Scheduler are randomly chosen and evaluated by an SMC experiment using the confidence bound for multiple schedulers [8]. The result displays both the minimum and the maximum probability.



This simple sampling has the disadvantage of allocating equal budget to all schedulers, regardless of their merit. To maximize the probability of finding an optimal scheduler with finite budget, Plasma Lab implements “smart sampling” algorithms [3,9], comprising three stages:

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

Note that smart hypothesis testing may quit at any stage if an individual scheduler is found to satisfy the hypothesis with required confidence or if individual schedulers do not satisfy the hypothesis with required confidence but the average of all schedulers satisfies the hypothesis.

This algorithm has several options, several of them being optional. The necessary parameters are Epsilon, Delta, that define the confidence bound, and Budget, that defines a fix number of simulations to perform at each stage. The other parameters have default values that can be substituted to optimize the performances of the algorithm:

- Max Budget, equal to Budget by default, is the number of simulations for the first stage.
- Reduction factor, by default 2, is the proportion of schedulers kept after each iteration.
- Initial probability allows to skip the first stage if not zero.
- Reward name, SPRT, Threshold, Alpha are used to compute the minimum or maximum value of a reward property, instead of a probability.

The image shows a configuration window for the 'Chernoff ND' algorithm. The 'Maximum' radio button is selected. The 'Budget' is set to 10000. An arrow points from the configuration window to a table with the following data:

#	Name	Max probability	Iterations	# Simulations
1	equal	0,5859375	10	89936

### 3.6 Rare Events Algorithms

Rare properties (i.e., with low probability) pose a problem for SMC because they are infrequently observed in simulations. Plasma Lab addresses this with the standard variance reduction techniques of importance sampling [4] and importance splitting [5,6,7].

Importance sampling works by weighting the probability distribution of the original system to favour the rare event. Since the weights are known, the correct result can be computed on the fly while simulating under the favourable importance sampling distribution. Importance sampling is implemented in Plasma Lab with the RML language. A language extension allows to add sampling parameters to the model to modify the rates of the actions. Then the cross entropy algorithm can be used to determine an optimal distribution of the sampling parameters.

Importance splitting decomposes a property with low probability into a product of higher conditional probabilities that are easier to estimate. It proceeds by estimating the probability of passing from one *level* to another, defined in Plasma Lab with respect to the range of a *score function* that maps states of the system  $\times$  property product automaton to values. The lowest level is the initial state. The highest level satisfies the property. The initial states of intermediate simulations are the terminal states of simulations reaching the previous level. The best performance is generally achieved with many levels of equal probability, requiring suitable score functions. Plasma Lab includes a “wizard” to construct *observers* in a reactive modules-like syntax from BLTL properties. The score function is defined within the observer and has access to all the variables of the system [7].

Plasma Lab implements a fixed level algorithm and an adaptive level algorithm [6,7]. The fixed level algorithm requires the user to define a monotonically increasing sequence of score values whose last value corresponds to satisfying the property. The adaptive algorithm finds optimal levels automatically and requires only the maximum score to be specified. Both algorithms estimate the probability of passing from one level to the next by the proportion of a constant number of simulations that reach the upper level from the lower. New simulations to replace those that failed to reach the upper level are started from states chosen uniformly at random from the terminal states of successful simulations. The overall estimate is the product of the estimates of going from one level to the next.

We consider a simple DTMC model that implements a chain of 100 states, with at each state a probability of 0.9 to pass to the next state, and a probability of 0.1 to exit to a state  $s=101$ . We want to estimate the probability to reach the final state  $s=100$ . We can use Monte Carlo with the BLTL property  $F<=\#100\ s=100$ . However the number of simulations must be very large to observe a significant number of successful traces. Otherwise we can use importance splitting with an observer that compute a score function: the score is equal to the last state reached in the trace.

---

```

dtmc
module chain
  s : [0..101] init 0;
  [] s<100 -> 0.90 : (s'=s+1)
    + 0.10 : (s'=101);
endmodule

observer chainObserver
  score : double init 0;
  decided : bool init false;
  [] s!=101 & score<s -> (score'=score+1);
  [] s>=100 -> (decided'=true);
endobserver

```

---

We compare in Table 5 the Monte Carlo algorithm, with 1'000'000 simulations, the fixed levels importance splitting algorithm, with 4 levels and 1'000 simulations, and the adaptive importance splitting algorithm with 1'000 simulations. Each experiment is performed 10 times and the table shows the average and the standard deviation of these 10 executions. The results show that the fixed levels algorithm produces an answer with a similar variance to Monte Carlo in a much faster time. The adaptive algorithm also produces a fast answer, and additionally improves the variance.

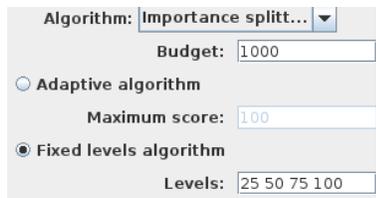


Fig. 4: Plasma Lab parameters for importance splitting.

---

	Probability	Time	Std. dev.
<b>Monte Carlo</b>	2.8E-5	7.93s	5.08E-6
<b>Fixed levels</b>	2.61E-5	0.58s	5.35E-6
<b>Adaptive</b>	2.56E-5	0.2s	1.88E-6

---

Fig. 5: Comparison between importance splitting algorithms and Monte Carlo

## 4 Plugin Development Tutorial

This tutorial explains how to develop a new simulator plugin for Plasma Lab. The sources of this tutorial can be downloaded on our documentation website ([http://plasma-lab.gforge.inria.fr/plasma\\_lab\\_doc/1.4.0/html/developer/tutorials/index.html](http://plasma-lab.gforge.inria.fr/plasma_lab_doc/1.4.0/html/developer/tutorials/index.html)).

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 is only used to illustrate the plugin creation and it has no stochastic property. We begin our simulator with the class declaration:

---

```
public class MySimulator extends AbstractModel
```

---

### 4.1 Factory

To load our simulator plugin in Plasma Lab we need a factory class that extends `AbstractModelFactory`. The factory class implements a new JSPF Plugin. Its plugin nature is declared using the annotation `@PluginImplementation` before the class declaration.

---

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

---

The main purpose of the factory is to instantiate a simulator or a checker without knowing its class. This is done by implementing the `createAbstractModel` methods to call the simulator constructor. For instance:

---

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

---

The factory also implements methods to identify a plugin by returning the name of the factory as it appears in the GUI menus (`getName`), a short textual description (`getDescription`), and a unique identifier (`getId`).

### 4.2 State and Identifiers

To create our new simulator we first need some companion objects to manipulate values and states. Identifiers are a shared objects to identify values (e.g. variables, constants) through different components of Plasma Lab. We create a new class `MyId` that implements the `InterfaceIdentifier` interface:

---

```
public class MyId implements Identifier {
    String name;

    public MyId(String name) {
        this.name = name;
    }
}
```

---

In our model we manipulate only two variables, the value and the time. We can declare them as static identifiers in the `MySimulator` class:

---

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

---

The 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 the two variables, time and value.

---

```
public class MyState implements InterfaceState {
    double value, time;

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

---

The class `MyState` additionally implements getters and setters to access and modify the values of the state, either through an `Identifier` object or through their name. For instance the following method returns the values the state according to the identifier:

---

```
public Double getValueOf(Identifier id) {
    if(id.equals(MySimulator.VALUEID))
        return value;
    else if(id.equals(MySimulator.TIMEID))
        return time;
    else
        throw new PlasmaRunException("Unknown identifier: "+id.getName());
}
```

---

### 4.3 Check for errors

The `checkForErrors` method is called before each experimentation/simulation and when modifying the content of the edition panel of the GUI. The purpose of this function is double: to detect any syntax error and to build the model before running it. Our `checkForErrors` method 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. Finally we create the initial state that will be used to initialize each trace.

---

```
@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("Not a valid command"));
    }
} catch (IOException e) {
    errors.add(new PlasmaException(e));
}
initialState = new MyState(0,0);
return !errors.isEmpty();
}

```

---

#### 4.4 New path

The `newPath` method initializes a new trace and returns the first state of the trace. `content` is a `String` inherited from `AbstractData` that 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` method.

```

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

```

---

#### 4.5 Simulate

The `simulate` method adds a new state to the trace. In our simulator we read the next character ('+' or '-') and we either add or subtract 1 to the value of the current state. We also add 1 to the time. Finally 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.

```

public InterfaceState simulate() throws PlasmaDeadlockException {
    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 PlasmaSimulationException(e);
    }
    return getCurrentState();
}

```

---

## References

1. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science* 2(5) (2006)
2. Boyer, B., Corre, K., Legay, A., Sedwards, S.: PLASMA-lab: A Flexible, Distributable Statistical Model Checking Library. In: *Proceedings of QEST*. LNCS, vol. 8054, pp. 160–164. Springer (2013)
3. D’Argenio, P., Legay, A., Sedwards, S., Traonouez, L.: Smart sampling for lightweight verification of markov decision processes. *STTT* 17(4), 469–484 (2015)
4. Jegourel, C., Legay, A., Sedwards, S.: Cross-entropy optimisation of importance sampling parameters for statistical model checking. In: *Proceedings of CAV*. LNCS, vol. 7358, pp. 327–342. Springer (2012)
5. Jegourel, C., Legay, A., Sedwards, S.: Importance Splitting for Statistical Model Checking Rare Properties. In: *Proceedings of CAV*. LNCS, vol. 8044, pp. 576–591. Springer (2013)
6. Jegourel, C., Legay, A., Sedwards, S.: An effective heuristic for adaptive importance splitting in statistical model checking. In: *Proceedings of ISoLA (2)*. LNCS, vol. 8803, pp. 143–159. Springer (2014)
7. Jegourel, C., Legay, A., Sedwards, S., Traonouez, L.: Distributed verification of rare properties using importance splitting observers. *ECEASST* 72 (2015)
8. Legay, A., Sedwards, S., Traonouez, L.: Scalable verification of markov decision processes. In: *SEFM Collocated Workshops*. LNCS, vol. 8938, pp. 350–362. Springer (2014)
9. Legay, A., Sedwards, S., Traonouez, L.: Estimating rewards & rare events in non-deterministic systems. *ECEASST* 72 (2015)
10. Okamoto, M.: Some inequalities relating to the partial sum of binomial probabilities. *Annals of the Institute of Statistical Mathematics* 10, 29–35 (1959)
11. Wald, A.: Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics* 16(2), 117–186 (1945)