Plasma Lab Statistical Model Checker: 
Architecture, Usage and Extension

Axel Legay and Louis-Marie Traonouez
Inria, Rennes, France

SOFSEM Tutorial – 19th January 2017
Outline

1. Introduction
   - Validation techniques
   - Statistical model-checking
   - Plasma Lab

2. Plasma Lab Architecture
   - Architecture
   - Available plugins
   - Rare events algorithms
   - SMC for nondeterminism

3. Plasma Lab Usage
   - Demo

4. Extension: Developing New Plugins
   - New simulator
   - New algorithm

5. Conclusion
1 Introduction
   • Validation techniques
   • Statistical model-checking
   • Plasma Lab

2 Plasma Lab Architecture
   • Architecture
   • Available plugins
   • Rare events algorithms
   • SMC for nondeterminism

3 Plasma Lab Usage
   • Demo

4 Extension: Developing New Plugins
   • New simulator
   • New algorithm

5 Conclusion
Validation Techniques

Examples:

- **Code reviewing**: rates are about 150 lines of code per hour. Inspecting and reviewing more than a few hundred lines of code per hour for critical software (such as safety critical embedded software) may be too fast to find errors.
- **Testing**: massively used, but faces the coverability problem and has problems with non determinism and quantities.
- ...

Another approach **Model Checking (Clarke, Emerson, Sifakis)**:

- Automated and exhaustive technique
- Relies on mathematical models for requirements and systems
- May suffer from state-space explosion.
Model Checking: Overview

- Requirements
- Formalizing
- Property Specification
- Model Checker
- System
- Modeling
- System model
- Temporal Logic
- Satisfied
- Violation + counterexample
- (Probabilistic) system

Introduction | Validation techniques
5 / 62
System S

Property $\Phi$

Parameters:
- Number of samples,
- Confidence,
- Precision

S Simulator
executable

$\Phi$ Monitor
executable

Produces an execution trace

SMC core
SSP, SPRT, PESTIM...
executable

Final verdict

Collects triggers

OK/KO local verdict
Easily parallelisable

Parameters:
Number of samples, Confidence, Precision

System S

S Simulator executable

Produces an execution trace

Property Φ

Φ Monitor executable

SMC core
SSP, SPRT, PESTIM...
executable

collects

triggers

OK/KO local verdict

Final verdict

Introduction
Statistical model-checking
Statistical Model Checking: Overview
(Youness, Larsen, Peyronnet)

Easily parallelisable

System S

Property Φ

Parameters:
Number of samples, Confidence, Precision

S Simulator executable

Φ Monitor executable

Produces an execution trace

SMC core executable

SSP, SPRT, PESTIM...

OK/KO local verdict

Final verdict

Less memory intensive than Model Checking

Final verdict
Statistical Model Checking: Overview
(Youness, Larsen, Peyronnet)

Parameters:
- Number of samples,
- Confidence,
- Precision

System S
- S Simulator
  - executable
- Produces an execution trace
- SMCCore
  - executable
  - SSP, SPRT, PESTIM...
- SMC core
- Final verdict
- OK/KO local verdict
- Collects
- Triggers

Property $\Phi$
- $\Phi$ Monitor
  - executable
- Approximately results
  - Given with Confidence Bounds
- Less memory intensive than Model Checking

Easily parallelisable
Combine formal methods with statistical techniques:

- Generate random simulation traces from a model/system.
- Check the traces against formal properties, *like the Bounded Linear Temporal Logic (BLTL).*
- Apply statistical algorithms to estimate probabilities, test hypotheses: *Monte-Carlo, SPRT, CUSUM, rare events techniques...*
Can be applied to any purely probabilistic system.

Lightweight algorithms that check bounded properties.

States are generated on-the-fly:
- No state-space is stored.
- Can be applied to infinite state systems.

Independent simulations can be divided on (massively) parallel computing architectures.
Requirement’s model in SMC

- **BLTL (Pnueli’77):**
  \[ \phi := \alpha | \phi \lor \phi | \phi \land \phi | \neg \phi | \Diamond \phi | \Diamond \leq t \phi | \Box \leq s \phi | \phi U \leq t \phi \]
  - \( \Diamond \): “next” operator
  - \( \Diamond \): “eventually” operator
  - \( \Box \): “always” operator
  - \( U \): “until” operator
  - \( \Diamond \leq t(x \geq 0) \): \( x \) will be eventually greater than 0 within \( t \) time units
BLTL (Pnueli’77):

\[ \phi := \alpha | \phi \lor \phi | \phi \land \phi | \neg \phi | \lozenge \phi | \lozenge^{\leq t} \phi | \square^{\leq t} \phi | \phi U^{\leq t} \phi \]

- \( \lozenge \): “next” operator
- \( \lozenge \): “eventually” operator
- \( \square \): “always” operator
- \( U \): “until” operator
- \( \lozenge^{\leq t}(x \geq 0) \): \( x \) will be eventually greater than 0 within \( t \) time units

Probabilistic BLTL:

- \( P_{\sim} \phi \) with \( \sim \in \{<, >, =\} \) and \( \theta \in [0, 1] \)
- \( P_{\leq 0.2}(\lozenge^{\leq t} \square^{\leq s}(x \geq 0)) \).
Quantitative Algorithm: Monte Carlo estimation

Estimate the probability $p$ of a BLTL property $\varphi$

$$\hat{p} \approx \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}(\omega_i \models \varphi)$$

- $N$ random simulations traces $\omega_i$
- Indicator function $\mathbb{1}(. )$
  - returns 1 if $\omega_i \models \varphi$
  - 0 otherwise
- The number $N$ of simulations can be determined using a confidence bound, like the Chernoff bound:

$$N \geq \frac{\ln(\frac{2}{\delta})}{2\varepsilon^2} \Rightarrow P(| \hat{p} - p | \geq \varepsilon) \leq \delta$$
Qualitative Algorithm: Hypothesis Testing (Wald’45)

Two hypothesis: $H_0 : p \leq \theta$, $H_1 : p > \theta$

Compute:

$$W = \prod_{i=1}^{m} \frac{P(\omega_i \models \varphi | p = \theta - \delta)}{P(\omega_i \models \varphi | p = \theta + \delta)} = \frac{(\theta - \delta)^{d_m}(1 - \theta + \delta)^{m-d_m}}{(\theta + \delta)^{d_m}(1 - \theta - \delta)^{m-d_m}},$$

where $d_m = \sum_{i=1}^{m} 1(\omega_i \models \varphi)$. 
Qualitative Algorithm: Hypothesis Testing (Wald’45)

Two hypothesis: \( H_0 : p \leq \theta, \ H_1 : p > \theta \)

Compute:

\[
W = \prod_{i=1}^{m} \frac{P(\omega_i \models \varphi \mid p = \theta - \delta)}{P(\omega_i \models \varphi \mid p = \theta + \delta)} = \frac{(\theta - \delta)^{d_m}(1 - \theta + \delta)^{m-d_m}}{(\theta + \delta)^{d_m}(1 - \theta - \delta)^{m-d_m}},
\]

where \( d_m = \sum_{i=1}^{m} \mathbb{1}(\omega_i \models \varphi) \).

Stop when:

- \( W \geq (1 - \beta)/\alpha : H_1 \) is accepted;
- \( W \leq \beta/(1 - \alpha) : H_0 \) is accepted.

\( \alpha \) is the probability to accept \( H_1 \) when \( H_0 \) holds. \( \beta \) is the probability to accept \( H_0 \) when \( H_1 \) holds.
A **PLA**tform for **Statistical Model Analysis**

Download at [https://project.inria.fr/plasma-lab/](https://project.inria.fr/plasma-lab/)

- A library of statistical model-checking algorithms
  
  *(Monte-Carlo, SPRT, rare events, CUSUM, nondeterminism,...)*

- Generic analysis for any runnable language or model

- Easily distributed other computation grid

- An API that allows modularity:
  
  *extendable with plugins to add new algorithms, new input languages.*

- Developed in Java 6
Outline

1. Introduction
   - Validation techniques
   - Statistical model-checking
   - Plasma Lab

2. Plasma Lab Architecture
   - Architecture
   - Available plugins
   - Rare events algorithms
   - SMC for nondeterminism

3. Plasma Lab Usage
   - Demo

4. Extension: Developing New Plugins
   - New simulator
   - New algorithm

5. Conclusion
Plasma Lab Architecture

Application-specific logics
Application-specific modeling languages
SMC algorithms
Distribution and management

Plasma Core

Controller

User Interface
External tools

Plasma Control

API
Parameters

Plasma Plugins

Algorithm
Checker
Simulator

SMC algorithms
Application-specific logics
Application-specific modeling languages

Results
Result
Request
Trace
**Plasma Lab API**

- **Plasma Lab UI**
- **Terminal**
  - Creates new experiments
  - Interfaces with PLASMA

**Model/Simulator**

- **Simulink**
- **Bio**
- **RML**

**Controller API**

- **PLASMA Lab UI**
- **MATLAB**

**PLASMA Lab**

**Requirement**

**PLASMA Lab plugins**

- **GCSL**
- **B-LTL**

**Plasma Lab Architecture**

- **Architecture**
Plasma Lab API

- Plasm Lab API
- Controller API
- Model/Simulator
- Requirement/Checker
- SMC Algorithms

- Decides if the property holds or not
- Property Monitoring during the simulations
- Estimates the probability
- Confidence Level set by the user
- Request Simulations for Checking

- Executes a model
- Builds execution traces

Plasma Lab Architecture
Simulators

Available simulator plugins:

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RML</td>
<td>Reactive Module Language: Input language of the tool Prism for Markov chains models</td>
</tr>
<tr>
<td>RML Adaptive</td>
<td>Extension of RML for adaptive systems</td>
</tr>
<tr>
<td>Bio</td>
<td>Biological language for writing chemical reactions</td>
</tr>
<tr>
<td>Matlab Session</td>
<td>Allows to control the simulator of Matlab/Simulink</td>
</tr>
<tr>
<td>SystemC</td>
<td>Simulation of SystemC models. 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.</td>
</tr>
<tr>
<td>Pi-ADL</td>
<td>Simulation of Architecture Description Language.</td>
</tr>
<tr>
<td>PTA</td>
<td>Probabilistic Timed Automata.</td>
</tr>
</tbody>
</table>
Available checker plugins:

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLTL</td>
<td>Bounded Linear Temporal Logic</td>
</tr>
<tr>
<td>ALTL</td>
<td>Adaptive Linear Temporal Logic</td>
</tr>
<tr>
<td></td>
<td>Extension of BLTL with new operators for adaptive systems.</td>
</tr>
<tr>
<td>GSCL</td>
<td>Goal and Contract Specification Language</td>
</tr>
<tr>
<td></td>
<td>A high level specification language for systems of systems.</td>
</tr>
<tr>
<td>Nested</td>
<td>BLTL checker enhanced with nested probability operator</td>
</tr>
<tr>
<td>RML Observer</td>
<td>A plugin that allows to write requirement as observers using a language</td>
</tr>
<tr>
<td></td>
<td>similar to RML. It is used to write rare properties.</td>
</tr>
</tbody>
</table>
## Algorithms

### Available algorithm plugins:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monte Carlo</td>
<td>Monte Carlo probability estimation with Chernoff-Hoeffding bound.</td>
</tr>
<tr>
<td>SPRT</td>
<td>Sequential Probability Ratio Test for hypothesis testing.</td>
</tr>
<tr>
<td>Importance splitting</td>
<td>Estimate the probability of rare events:</td>
</tr>
<tr>
<td></td>
<td>- Decompose a requirement with low probability into a product of higher</td>
</tr>
<tr>
<td></td>
<td>conditional probabilities that are easier to estimate.</td>
</tr>
<tr>
<td>Importance sampling with cross entropy</td>
<td>Estimate the probability of rare events:</td>
</tr>
<tr>
<td></td>
<td>- Weight the probability distribution of the original system to favour the</td>
</tr>
<tr>
<td></td>
<td>rare event.</td>
</tr>
<tr>
<td></td>
<td>- Use cross entropy, to determine an optimal weighted distribution.</td>
</tr>
<tr>
<td>SMC for non-deterministic models</td>
<td>“Smart sampling” algorithms to estimate minimum and maximum probabilities</td>
</tr>
<tr>
<td></td>
<td>in non deterministic models.</td>
</tr>
<tr>
<td>CUSUM</td>
<td>Change detection problem:</td>
</tr>
<tr>
<td></td>
<td>performs runtime monitoring of a single trace.</td>
</tr>
</tbody>
</table>
Model Language: RML

Input language of the model-checker PRISM:
- Textual language for modeling DTMC, CTMC, MDP, PTA
- Guarded commands transitions:
  \[ \text{[synchro] guard} \rightarrow \text{rate1:}(\text{action1}) + \text{rate2:}(\text{action2}) \]
- System description via modules renaming
dtmc

formula lfree = p2>=0 & p2<=4 | p2=6 | p2=10;
formula rfree = p3>=0 & p3<=3 | p3=5 | p3=7;
module phil1
    p1: [0..10];
    [] p1=0  ->  0.2 : (p1'=0) + 0.8 : (p1'=1);
    [] p1=1  ->  0.5 : (p1'=2) + 0.5 : (p1'=3);
    [] p1=2 & lfree  -> (p1'=4);
    [] p1=2 & !lfree  -> (p1'=2);
    [] p1=3 & rfree  -> (p1'=5);
    [] p1=3 & !rfree  -> (p1'=3);
    [] p1=4 & rfree  -> (p1'=8);
    [] p1=4 & !rfree  -> (p1'=6);
    [] p1=5 & lfree  -> (p1'=8);
    [] p1=5 & !lfree  -> (p1'=7);
    [] p1=6  -> (p1'=1);
    [] p1=7  -> (p1'=1);
    [] p1=8  -> (p1'=9);
    [] p1=9  -> (p1'=10);
    [] p1=10  -> (p1'=0);
endmodule
module phil2=phil1 [p1=p2, p2=p3, p3=p1]  endmodule
module phil3=phil1 [p1=p3, p2=p1, p3=p2]  endmodule

label "hungry" = ((p1>0)&(p1<8))|((p2>0)&(p2<8))|((p3>0)&(p3<8));
label "eat" = ((p1>=8)&(p1<9))|((p2>=8)&(p2<9))|((p3>=8)&(p3<9));
Textual language for biological models:

- Write chemicals reactions with CTMC semantics:
  \[ \text{product1} + \text{product2} \text{ rate} \rightarrow \text{product3} + \text{product4} \]

- Use Gillespie algorithm for simulating biological models:
  The time and probability of a reaction depends on its rate and the number of species.

<table>
<thead>
<tr>
<th>#</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0</td>
<td>1000.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>1.4033595075152232E-6</td>
<td>999.0</td>
<td>999.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>3.1610719924273105E-6</td>
<td>998.0</td>
<td>998.0</td>
<td>2.0</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>3.489499125855082E-6</td>
<td>997.0</td>
<td>997.0</td>
<td>3.0</td>
<td>0.0</td>
</tr>
<tr>
<td>5</td>
<td>4.132233001983635E-6</td>
<td>996.0</td>
<td>996.0</td>
<td>4.0</td>
<td>0.0</td>
</tr>
<tr>
<td>6</td>
<td>6.13913509039687E-6</td>
<td>995.0</td>
<td>995.0</td>
<td>5.0</td>
<td>1.0</td>
</tr>
<tr>
<td>7</td>
<td>7.55365258629189E-6</td>
<td>994.0</td>
<td>994.0</td>
<td>6.0</td>
<td>1.0</td>
</tr>
<tr>
<td>8</td>
<td>9.373545684697474E-6</td>
<td>993.0</td>
<td>993.0</td>
<td>7.0</td>
<td>1.0</td>
</tr>
<tr>
<td>9</td>
<td>1.0584090810831553E-5</td>
<td>992.0</td>
<td>992.0</td>
<td>8.0</td>
<td>1.0</td>
</tr>
<tr>
<td>10</td>
<td>1.0856701877068095E-5</td>
<td>991.0</td>
<td>991.0</td>
<td>9.0</td>
<td>1.0</td>
</tr>
<tr>
<td>11</td>
<td>1.2767574877520022E-5</td>
<td>990.0</td>
<td>990.0</td>
<td>10.0</td>
<td>1.0</td>
</tr>
<tr>
<td>12</td>
<td>1.4486346791495854E-5</td>
<td>989.0</td>
<td>989.0</td>
<td>10.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**species** A=1000, B=1000, C, D, E

A + B → C
C 10000 → D
D → E

Plasma Lab Architecture

Available plugins

22 / 62
Properties Language: BLTL

Checker for the Bounded Linear Temporal Logic:

- **Syntax:**
  - F: Eventually
  - G: Always
  - U: Until
  - X: Next

- Generate states “on demand” depending on the bounds of the property.

**Examples**

\[
F \leq \#50\text{"eat"}
\]

\[
F \leq \#1000(\text{"hungry"} \& (X \leq \#1F \leq \#1000\text{"eat"}))
\]
Rare Events

Goal: estimate the probability of rare events (e.g. $10^{-6}$ probability).

- Simulation techniques require a very large number of simulations to generate rare events.
- Confidence interval based on absolute error, like in the Monte Carlo technique, are not relevant.
Goal: Increase the probability of generating traces that satisfy a rare event property.
Rare Events Simulation Algorithms

Goal: Increase the probability of generating traces that satisfy a rare event property.

Two approaches:

1. Split a rare event property into a sequence of less rare properties:
   ⇒ Importance splitting technique
Rare Events Simulation Algorithms

**Goal:** Increase the probability of generating traces that satisfy a rare event property.

Two approaches:

1. Split a rare event property into a sequence of less rare properties:
   ⇒ Importance splitting technique

2. Modify the probability distribution of the model with a weight in order to generate more often the rare events:
   ⇒ Importance sampling technique
Importance Splitting

Divide a difficult estimation into a product of easier estimations.

- Ideally requires properties of form:
  \[ \varphi = \varphi_n \Rightarrow \varphi_{n-1} \Rightarrow \cdots \Rightarrow \varphi_1 \Rightarrow \varphi_0 \]

- Probability is then of form:
  \[ P(\varphi) = \prod_{i=1}^{n} P(\varphi_i \mid \varphi_{i-1}) \]

- Works best with many levels of equal probability.
Logical decomposition are often too coarse:
⇒ uneven probabilities

Use score function as abstract level of simulation:
⇒ maps state of model-property product automaton to value:

\[ S(\omega) > S(\omega') \iff P(\omega \models \varphi) > P(\omega' \models \varphi) \]

Probability is of form:

\[
P(\varphi) = \prod_{i=1}^{n} P(s \geq s_i \mid s \geq s_{i-1})
\]
How to write score functions

- **Logical decomposition:**
  
  If \( \varphi = \varphi_n \Rightarrow \varphi_{n-1} \Rightarrow \cdots \Rightarrow \varphi_1 \Rightarrow \varphi_0 \),
  
  Then \( S(\omega) > S(\omega') \iff P(\omega \models \varphi_{i+1}) > P(\omega' \models \varphi_i) \)
  
  Often too coarse and uneven.

- **Natural decomposition:**
  
  incremental counters, continuous quantities

- **Heuristics:**
  
  Partition that “makes sense”
  
  e.g. shorter traces have higher score
Implementing Score Functions in Plasma Lab

Observers

// G<=#420 !"elected"

observer obs1 // atomic
  obs1_output : bool;
  [] true → (obs1_output’ = ! "elected");
endobserver

observer obs0 // globally
  obs0_output : bool;
  decided : bool;
  score : [0..420];
  [] !decided & obs1_output & score < 420 → (score’ = score + 1);
  [] !decided & !obs1_output → (decided’ = true) & (obs0_output’ = false);
  [] !decided & obs1_output & score = 420 → (decided’ = true) & (obs0_output’ = true);
endobserver

¬elected ∧ score < 420, score ← score + 1

score ← 0

¬elected ∧ score = 420

¬elected

elected

✓
Fixed Levels Algorithm

\[ P(\varphi_1 | \varphi_0) = \frac{3}{5} \]

Advantage: Low memory overhead (only stores terminal states)
Fixed Levels Algorithm

$P(\varphi_2 | \varphi_1) = \frac{3}{5}$

$P(\varphi_1 | \varphi_0) = \frac{3}{5}$
Fixed Levels Algorithm

\[ P(\varphi) = \left(\frac{3}{5}\right)^3 = 0.216 \]

\[ P(\varphi_3|\varphi_2) = \frac{3}{5} \]

\[ P(\varphi_2|\varphi_1) = \frac{3}{5} \]

\[ P(\varphi_1|\varphi_0) = \frac{3}{5} \]
**Fixed Levels Algorithm**

\[ P(\varphi) = \frac{3}{5}^3 = 0.216 \]

\[ P(\varphi_3 | \varphi_2) = \frac{3}{5} \]

\[ P(\varphi_2 | \varphi_1) = \frac{3}{5} \]

\[ P(\varphi_1 | \varphi_0) = \frac{3}{5} \]

**Advantage:** Low memory overhead (only stores terminal states)
Adaptive Levels Algorithm

Score

Level $\phi$

Level 0

Simulation time

bound
Adaptive Levels Algorithm

Advantage: Optimized algorithm
But: High memory overhead (stores traces)

Plasma Lab Architecture

Rare events algorithms
Adaptive Levels Algorithm

- Advantage: Optimized algorithm
- But: High memory overhead (stores traces)

Plasma Lab Architecture

Rare events algorithms
Adaptive Levels Algorithm

**Advantage:** Optimized algorithm

**But:** High memory overhead (stores traces)
Importance Sampling

- Weight system measure to make property less rare.
- Numerically compensate for weights.

\[
P_F(\varphi) = \int_{\Omega} \mathbb{1}_{\varphi}(\omega) \frac{dF}{dG} dG
\]

\[
P_F(\varphi) \approx \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_{\varphi}(\omega_i) \frac{dF(\omega_i)}{dG(\omega_i)}
\]

Implemented in Plasma Lab for the RML language and PTA.
Fewer traces fail to satisfy $\varphi$

Failures are rare, but the results will be mostly underestimated
Optimal sampling distribution \( dG = \frac{1}{P_F(\varphi)} dF \)

Min-cross entropy parameterized distribution
Markov Decision Processes (MDP)

\( V \) : set of integer variables,
\( S \) : set of states, valuations of \( V \),
\( A \) : set of nondeterministic actions guarded by predicates over \( V \),
\( P : S \times A \times S \to [0, 1] \), a probability function,
\( U : S \times A \times S \to 2^{V \times \mathbb{Z}} \), an update function,

Memoryless scheduler: \( S \to A \)
History-dependent scheduler: \( S^+ \to A \)
MDP Schedulers

\[ P(\Diamond \phi) = p \ast p \]

\[ P(\Diamond \phi) = 0 \]

\[ P(\Diamond \phi) = p \ast q \]
Lightweight Schedulers for SMC

- Hash function + PRNG deterministically resolves nondeterminism in all states.
- Applicable to memoryless and history-dependent schedulers.
SMC for MDP

Goal: Estimate minimum or maximum probabilities.
SMC for MDP

**Goal:** Estimate minimum or maximum probabilities.

**Algorithm 1:** Direct sampling of the scheduler space

1. Fix a number of schedulers to explore.
2. Evaluate with classical SMC algorithms the DTMC induced by each scheduler.

Confidence with multiple estimates:
e.g., $N \geq \left( \ln 2 - \ln \left(1 - \frac{M}{\sqrt{1 - \delta}}\right) \right) / (2\varepsilon^2)$
SMC for MDP

Goal: Estimate minimum or maximum probabilities.

Algorithm 1: Direct sampling of the scheduler space
1. Fix a number of schedulers to explore.
2. Evaluate with classical SMC algorithms the DTMC induced by each scheduler.

Confidence with multiple estimates:
e.g., $N \geq \left( \ln 2 - \ln \left( 1 - \frac{M}{\sqrt{1 - \delta}} \right) \right) / (2\varepsilon^2)$

Algorithm 2: Smart sampling
allocate more simulations to the best schedulers
Iterative algorithm with a fixed per-iteration budget $B = NM$ of simulations

- $M$ schedulers
- $N$ samples per scheduler
Smart Sampling SMC Algorithms

Iterative algorithm with a fixed per-iteration budget $B = NM$ of simulations

- $M$ schedulers
- $N$ samples per scheduler

1. Estimate the probability of optimal scheduler ($p_s$) and probability of finding a near-optimal scheduler ($p_g$) using $N_1 = M_1 = \sqrt{B}$
Iterative algorithm with a fixed per-iteration budget $B = NM$ of simulations

- $M$ schedulers
- $N$ samples per scheduler

1. Estimate the probability of optimal scheduler ($p_s$) and probability of finding a near-optimal scheduler ($p_g$) using $N_1 = M_1 = \sqrt{B}$

2. Find $N_2M_2 = B$ to approximately maximize $(1 - (1 - p_g)^M)(1 - (1 - p_s)^N)$
   Sample $M_2$ schedulers and test them with $N_2$ simulations each. Keep $M_3$ candidate schedulers that satisfy the property.
Smart Sampling SMC Algorithms

Iterative algorithm with a fixed per-iteration budget $B = NM$ of simulations

- $M$ schedulers
- $N$ samples per scheduler

1. Estimate the probability of optimal scheduler ($p_s$) and probability of finding a near-optimal scheduler ($p_g$) using $N_1 = M_1 = \sqrt{B}$

2. Find $N_2 M_2 = B$ to approximately maximize $(1 - (1 - p_g)^M)(1 - (1 - p_s)^N)$
   Sample $M_2$ schedulers and test them with $N_2$ simulations each. Keep $M_3$ candidate schedulers that satisfy the property.

3. Iteratively refine the set of candidates, maintaining $N_i M_i = B$
   - test $M_i$ schedulers with $N_i$ samples and keep best half.
Distributing SMC Experiments

Plasma Lab API provides generic methods to define distributed algorithms:

**Architecture:**

- Implemented with Restlet
- Dedicated client interface: Plasma Lab Service
Outline

1. Introduction
   - Validation techniques
   - Statistical model-checking
   - Plasma Lab

2. Plasma Lab Architecture
   - Architecture
   - Available plugins
   - Rare events algorithms
   - SMC for nondeterminism

3. Plasma Lab Usage
   - Demo

4. Extension: Developing New Plugins
   - New simulator
   - New algorithm

5. Conclusion
1. Download at https://project.inria.fr/plasma-lab/download/

2. Extract

3. Run ./plasmagui.sh launch or ./plasmagui.bat launch
Example: the Dining Philosophers Problem

What we should obtain:

- 150 philosophers
- property of interest: $\phi = \phi_5 = F^{30}$ (Phil i eat)
- $\gamma \approx 1.59 \times 10^{-6}$
Developing New Plugins

Plugins are of 3 types: **Simulator**, **Checker** or **Algorithm**.

Each plugin has a **factory** that is loaded dynamically by Plasma Lab.

**New Simulator**
- `newPath()`
  Start a simulation
- `simulate()`
  Simulate one step

**New Checker**
- `check(path)`
  Check a trace until the property is decided
- Return the result

**New Algorithm**
- `run()`
  Run the algorithm
  Send the results

Tutorial manual and sources at:
http://plasma-lab.gforge.inria.fr/plasma_lab_doc/1.4.0/html/developer/tutorials/index.html
Developing a New Simulator

Simple illustrative language

- Succession of + or –
- Initial state is 0
- + adds 1
- – subtracts 1
- +++-- produces the trace 0 1 2 3 2 1
Developing a New Simulator

Main classes:

**MySimulator**
- Extends AbstractModel
- newPath()
- simulate()

**MySimulatorFactory**
- Extends AbstractModelFactory
- createAbstractModel()

Companion objects

- **MyId** extends InterfaceIdentifier
- **MyState** extends InterfaceState
Factory

- Defines the new plugin loaded by Plasma Lab
- Create a simulator without knowing its class.

Implement the simulator construction methods

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

The factory also implements methods to identify a plugin:

- getName,
- getDescription
- getId.
Factory declaration

- Extends `AbstractModelFactory`.
- Implements a new JSPF Plugin (Java Simple Plugin Framework).
- Its plugin nature is declared with the annotation `@PluginImplementation`.

Create the simulator factory

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

Shared objects to identify values (e.g. variables, constants).
In our model we store two values:

- An integer value whose name is X
- A times step whose name is #

Create a new type of identifier

```java
public class MyId implements InterfaceIdenfifier {
    String name;
    public MyId(String name) {
        this.name = name;
    }
}
```

Instantiate the 2 identifiers

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

States are used to store the values of the model’s execution:

- Extends the `InterfaceState` interface.
- Implements methods to access the values.

In our new simulator states have only two variables: time and value.

Create a state class

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

    public MyState(double value, double time) {
        this.value = value;
        this.time = time;
    }
}
```
States: getters and setters

Access and modify the values of the state:
- either through an `InterfaceIdentifier` object,
- or through their name.

Implement `getValueOf`

```java
public Double getValueOf(InterfaceIdentifier id) {
    if (id.equals(MySimulator.VALUEID))
        return value;
    else if (id.equals(MySimulator.TIMEID))
        return time;
    else
        throw new PlasmaRunException("Unknown identifier: "+id.getName());
}
```
Simulator and models are the same object:

Create a new simulator class

public class MySimulator extends AbstractModel

Our simulator has the following attributes:

- **String content**: content of the model
- **List<Interface> trace**: current trace
- **BufferedReader br**: read the model character by character.
CheckForErrors method

- Called before each experimentation/simulation.
- Used to detect any syntax error.
- Initialize the model: we create an initial state.

**Implements checkForErrors in MySimulator**

```java
public boolean checkForErrors() {
    errors.clear();
    ByteArrayInputStream 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();
}
```
NewPath method

- Initializes a new trace with the initial state.
- Initializes the BufferedReader to read the content of the model.
- Returns the first state of the trace.

**Implements** `newPath` in `MySimulator`

```java
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 method

Create a new state and add it to the trace:

1. Read the next character
2. Get the current values of VALUEID and TIMEID
3. Create the new state according to the character read
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();
}
Developing a New Algorithm

Main classes:

<table>
<thead>
<tr>
<th>MyAlgorithm</th>
<th>MyAlgorithmFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implement InterfaceAlgorithmScheduler</td>
<td>Implements InterfaceAlgorithmFactory</td>
</tr>
<tr>
<td>run(): execute the algorithm, send the results.</td>
<td>createScheduler(): parse the parameters, construct a scheduler.</td>
</tr>
</tbody>
</table>

Companion object

- MySMCResult implements SMCResult: container for the results values.
The run method

public void run() {
    listener.notifyAlgorithmStarted("MyMC");
    double res = 0.0;
    for (int i = 0; i < nbSimu; i++) {
        InterfaceState path = model.newPath();
        res += requirement.check(path);
    }
    listener.notifyAlgorithmCompleted("MyMC");
    listener.publishResults("MyMC", new SMCR Result(res/nbSimu));
}

- model.newPath(): Start a new trace.
- requirement.check(path): Check the trace and collect the results.
- listener.publishResults: Send the results to the GUI/CLI.

New simple Monte Carlo
Developing a New Distributed Algorithm

Main classes:

**MyScheduler**
- `run()`: send orders to the workers, collect partial results, send final results.

**MyWorker**
- `connect()`: receive orders, execute the work, send partial results.

**MyAlgorithmFactory**
- `createScheduler()`, `createWorker()`

Companion objects
- **MyOrder**: work order send to the workers.
- **MyResult**: partial results send by the workers.
- **MySMCResult**.
Outline

1 Introduction
   1. Validation techniques
   2. Statistical model-checking
   3. Plasma Lab

2 Plasma Lab Architecture
   1. Architecture
   2. Available plugins
   3. Rare events algorithms
   4. SMC for nondeterminism

3 Plasma Lab Usage
   1. Demo

4 Extension: Developing New Plugins
   1. New simulator
   2. New algorithm

5 Conclusion
We have presented Plasma Lab, a modular statistical model-checker.

Plasma Lab includes:

- several input languages,
- several advanced SMC algorithms,
- an API to control experiments, distribute simulations, add new plugins.

Add your own language/algorithm to Plasma Lab!