

WVAISE 2012

MODELING AND VERIFYING DISTRIBUTED SYSTEMS WITH PETRI NETS

TUTORIAL - PRACTICAL WORK

NOVEMBER 12, 2012

Souheib Baarir

LIP6

Université Paris Ouest Nanterre

200 avenue de la République

92000 Nanterre

France

Souheib.Barrir@lip6.fr

Fabrice Kordon

LIP6

Université P. & M. Curie

4 Place Jussieu

75005 Paris

France

Fabrice.Kordon@lip6.fr



Contents

1	Introduction	1
2	Installing the environment	1
3	First Practice – Simple swimming pool	3
4	Second Practice – a protocol for the swimming pool	4
5	Third Practice – couples of clients/servers	5
A	Syntax for CTL fomulæ in PNXDD	6

1 Introduction

This document provide all the required information to perform the practical work planned for the “Modeling and Verifying Distributed Systems with Petri Nets” tutorial.

We first present the *CosyVerif*¹ environment, jointly developed by LIP6, LIPN and LSV, three laboratories in le de France where an strong verification activity in formal methods take place.

Then, the two parts of the pratical work are presented.

2 Installing the environment

CosyVerif is a generic verification environment based on a client/serveur approach (see figure 1). It is composed of a graphical user interface, *Coloane*, and an integration platform that allows one to plyg his own tools, *Alligator*. Both communicate via web-services. *Coloane* can be replaced by any client supporting the web services required to operate the tools integrated in the tool server.



Figure 1: Architecture of teh *CosyVerif* Environment.

So, you must operate both parts of the environment to use *CosyVerif*. The server and the clients may run on the same machine, this is the case for this practical work that you will perform on your own machine. There is a public environment reachable from the user interface but it requires a good internet connection.

As in the web site (download section), we provide a distribution for the graphical user interface and tool server. You must get these two files prior to any installation.

The tool server. The tool server is distributed as an image disk running on a virtual machine (in the *Alligator.zip* archive). VirtualBox should be previously istalled in the tool server machine (here, your computer). When unpacking the archive, you get a directory with to files:

- *Alligator.vbox*: this is the virtual box configuration file. Typically, double clicking on this file should launch VirtualBox with the appropriate disk image,
- *Alligator.vdi*: this is the disk image containing the tool server.

When VirtualBox has launched your tool server, the following sentences should be displayed:

```
Starting CosyVerif server (Alligator) on http://10.0.2.15:9000/
Startig Alligator server
```

The tool server is now ready to be contacter by the User Interface.

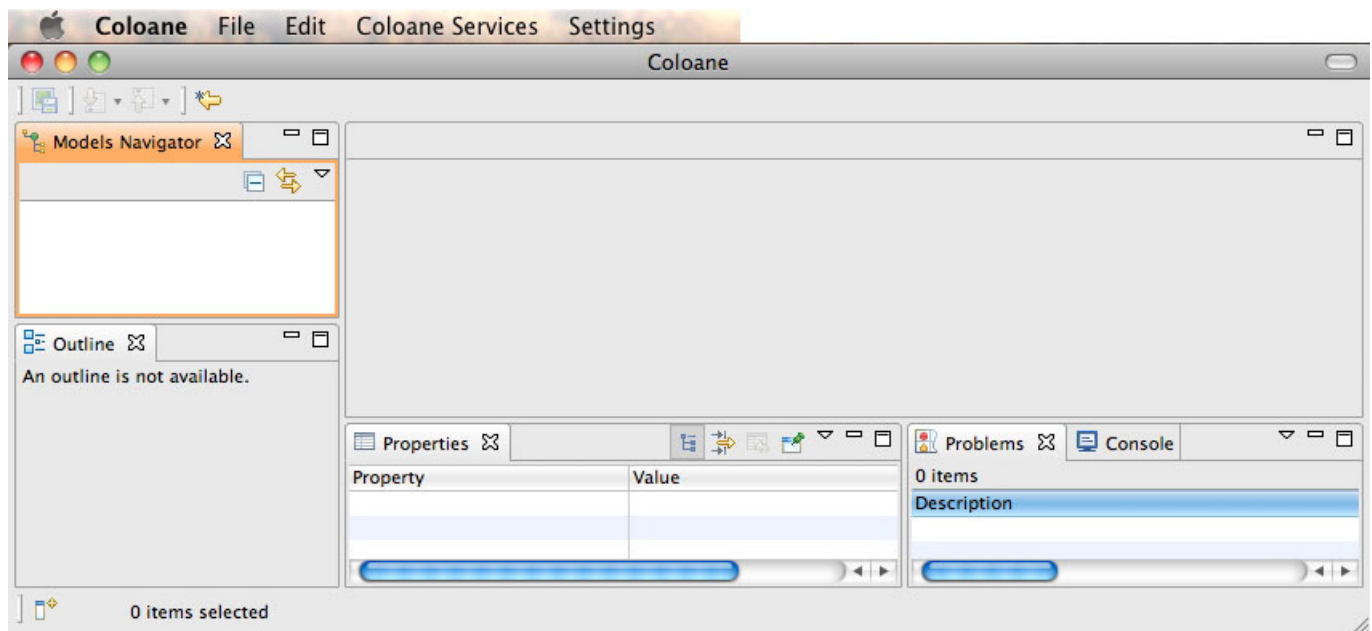


Figure 2: The *Coloane* User Interface (here under MacOS).

Coloane, the Graphical User Interface². You must select your distribution (among Linux, MacOS, Windows, both 32 and 64 bits). It is built on top of Eclipse but distributed as a standalone executable file.

Once the file uncompressed, you get a directory containing an executable file. Run it to access a working window similar to the one presented in figure 2) that is typical of an Eclipse working environment.

On the left, you find navigators for files and in the model. Then, in the center, you have the working zone, above the property window (left) and the console (right).

The “ColoaneServices” menu allows you to access to the services provided by the tool server. But you need first to get into the Settings menu to set-up preferences. You need to go in the “Coloane” section, then select the “Alligator Preference” subsection. If you use the public environment, nothing has to be changed. Otherwise, you must specify the URL of your environment:

`http://your server/servicemanager`

In the configuration of this tutorial, “your server” is “localhost:9000”. Click on “Apply” to have your preferences updated.

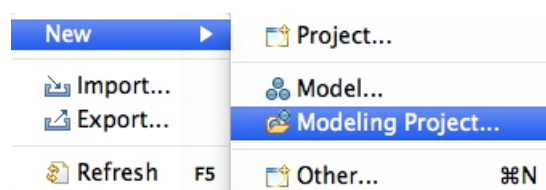


Figure 3: creating a modeling project

¹See <http://cosyverif.org> for more information (documentation, download the environment, etc.).

²The tool server is reachable by means of web services. Thus, any program able to invoke *Alligator* with its WS protocol can be substituted to *Coloane*.

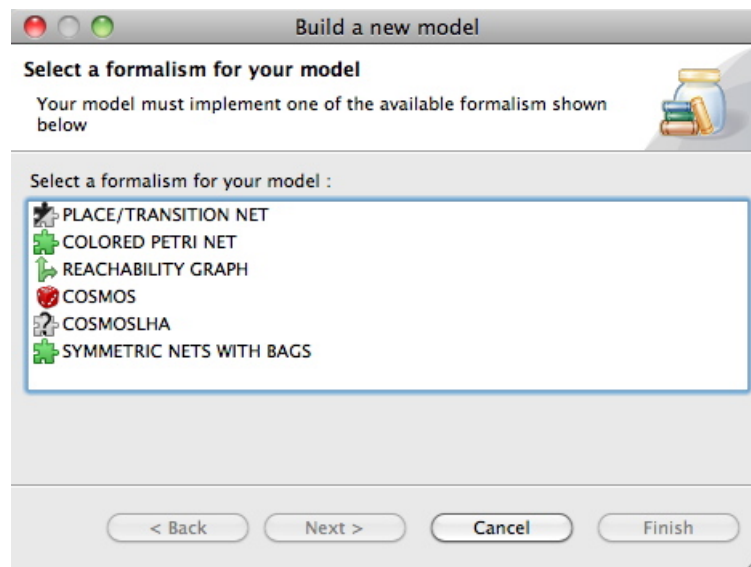


Figure 4: selecting a formalism

You must create a modeling project in your Coloane repository. To do so, control-click on the file navigator and select “New/Modeling Project...” as shown in figure 3 and provide a name to your project as for any Eclipse project.

To create a model, you repeat this operation but you need to select “New/Model...”. You then have to select the formalism to be used for modeling among the choice provided by *Coloane* (see figure 4). For this practical work, only two formalisms are used: “Place/Transition nets” and “Colored Petri net”. Select your formalism, enter a model name (a default one is proposed) and a palette to create Petri net objects appears in the modeling zone.

You are now ready to go on the practical work for this tutorial. You may design a Petri net model, affect name to places and transitions, marking to places and values to arcs (using the property window). Once your model is debugged, you may launch a service from the tool server via the “Coloane Services” menu.

3 First Practice – Simple swimming pool

This first practical sections aims to let you specify a simple system with P/T nets.

Question 1: swimming pool, first approach. One wants to model the behavior of a swimming pool. In this system, users behave using the following protocol:

1. enter in the swimming pool building,
2. get the key of a cabin they keep until they leave the swimming pool,
3. enter the cabin and change their clothes against a swimming suit,
4. get into the swimming pool, have fun and swim,
5. get back to the cabin, dry and get dressed,
6. return the cabin key,

7. exit the building.

You must model this system using a P/T net. For initial values of the system, you may consider 10 users and 4 cabins.

Question 2: a little bit of reachability analysis. Use reachability formulæ to logically identify the states where a maximum of users can stay in the swimming pool.

Once you have determined the maximum number of people in the swimming pool for this initial marking, please extrapolate this value for any configuration with u users and c cabins.

4 Second Practice – a protocol for the swimming pool

Question 1: a Swimming Pool Protocol. A new director is hired and suggests a new protocol to avoid the limitation previously observed (first practice). Now, a user must behave as follows:

1. enter in the swimming pool building,
2. get a cabin key,
3. get a bag for his clothes, it will keep this bag until he gets out.
4. enter the cabin, change clothes for his swimming suit,
5. return the cabin key to the office,
6. drop the bag with his clothes in the office too,
7. get into the swimming pool, have fun and swim,
8. request for a cabin key, then request for the bag containing his clothes,
9. get back to the cabin, dry and get dressed,
10. give back the key at the office,
11. give back the bag at the office,
12. exit the building.

You must now model this new protocol. For the initial value, we suggest 4 cabins, 12 users and 8 bags for clothes.

Question 2: analysis of the new swimming pool capacity. Evaluate the capacity of the swimming pool for the suggested initial marking. Please extrapolate to compute this capacity based on the following parameters: c (the number of cabins), b (the number of bags) and u (the number of people in the system).

Question 3: deadlock-free property. The director wants to check if any user entering in the building can leave after some finite time spent in the pool. Write the CTL formula that expresses this

Question 4: Evaluating the property. Is this property true? If not:

- why?
- what must be changed in the model to avoid this?

5 Third Practice – couples of clients/servers

We study a simple client/server problem.

Question 1: modeling the problem. Let us consider a set of clients and servers. Each client and server is identified by its id.

Each client sends a query message with its id to a dedicated server (the one with the same id). Servers only get message having their Id from the network. Once they received a message, they send an acknowledge implemented as an event with no semantics to the clients.

The following hypotheses are made on the network:

- it is reliable,
- it does not preserve the order of messages.

Question 2: analysis of messages. is it possible to have twice a on the network a query holding the same id? express this as a colored reachability formula. Check this formula with PNDD. What can you conclude?

Question 3: corrected model. Correct the previous specification and check again your formula on it.

A Syntax for CTL fomulæ in PNXDD

PNXDD follows the syntax of the VIS model checker.

An entire formula should be followed by a semicolon. All text from # to the end of a line is treated as a comment.

Atomic Propositions. TRUE, FALSE, and var-name=value are CTL formulas, where var-name is the full hierarchical name of a variable , and value is a legal value in the domain of the variable.

var-name1 == var-name2 is the atomic formula that is true if var-name1 has the same value as var-name2. Currently it can be used only in the Boolean domain (it cannot be used for variables of enumerated types).

Places are encoded as variables and their marking can be compared. The following expression compares the marking of $place_1$ with the one of $place_2$.

$$place_1 = place_2$$

The marking can be compared to a constant (number of tokens) as follows. The following formula checks there are 3 tokens in $place$

$$place = 3$$

Only equality can be tested with PNXDD at this stage.

Boolean and Logical Operators. Atomic propositions can be combined into more complex ones thanks to the following operators:

- *: boolean and,
- +: boolean or,
- ^: boolean xor,
- !: boolean negation,
- ->: implication,
- <->: equivalence.

Composed atomic expressions can be parenthesized.

CTL Operators. the following CTL operators are available (f and g are atomic propositions):

- AG f : f is always true,
- AF f : f is always verified in the future,
- AX f : f is always true at the next step,
- EG f : f is true sometimes,
- EF f : f is true in some future,
- EX f : f is true in some of the next step,
- A(f U g): f is always true until g becomes true,

- $E(f \text{ U } g)$: sometimes, f remains true until g becomes true.

Operator Precedence. Operators are evaluated using the following priority (1 is the highest):

1. $!$
2. $AG, AF, AX, EG, EF, EX,$
3. $*$,
4. $+$,
5. $^{\wedge}$,
6. \leftrightarrow ,
7. \rightarrow ,
8. U .