UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Magistrale in Informatica

Exam Document

FORMAL METHODS FOR SYSTEMS VERIFICATION

MASSIMO NOCENTINI

Professors: *Michele Loreti, Mieke Massink*

Anno Accademico 2012-2013

Massimo Nocentini: *Formal methods for systems verification,* Corso di Laurea Magistrale in Informatica, © Anno Accademico 2012-2013

CONTENTS

GOAL

This document collects my work on the course *Formal methods for system verification*.

Some useful web location

The source of the exercises is:
http://www-i2.informatik.rwth-aachen.de/i2/mvps11/

This document is hosted in the following `git` repository:
https://github.com/massimo-nocentini/metodi-formali-verifica-sistemi

LICENCES

Text contents

Sources

1 QUANTITATIVE PART

1.1 BASIC DEFINITIONS

1.1.1 *Stochastic Processes*

A *stochastic process* p is a $\mathcal{T}$-indexed family of random variables X_i, formally $p = \{X_i : i \in \mathcal{T}\}$, in particular p is called *discrete* if $\mathcal{T} = \mathbb{N}$, otherwise p is called *continuous* if $\mathcal{T} = \mathbb{R}$.

1.1.2 *Discrete Time Markov Chains*

A *Discrete Time Markov Chain* (DTMC) is a discrete stochastic process such that, given Ω a sample space, $X_i : \Omega \to \mathbb{N}, \forall i \in \mathbb{N}$.

1.1.3 *Continuous Time Markov Chains*

A *Continuous Time Markov Chain* (CTMC) is a continuous stochastic process such that, given Ω a sample space, $X_i : \Omega \to \mathbb{N}, \forall i \in \mathbb{R}$.

1.2 HERMAN'S STABILIZATION ALGORITHM

This paper study an algorithm about self-stabilization of fault-tolerant systems in a distributed environment. We can abstract a fault-tolerant system with a network of processes and the goal is to build a protocol which, applied to an initial configuration i of the network, produces a new configuration c such that:

- c satisfies some property of interest;
- the protocol requires a *finite* number of steps to produce c;
- there is no intervention of outside objects in order to produce c.

In the following section we state formally the problem described above and the Herman algorithm under study. We've used the article [KNP12] to make our abstractions.

1.2.1 *Model*

Herman protocol [Her90] is applied to a network of processes p_i where $i \in \{1, \dots, n\}$, structured in an oriented ring and ordered anticlockwise. This protocol is synchronous, that is actions taken by each process p_i happen simultaneously, no interleaving are present. Following the presentation given by Herman, it is possible to represent each process p_i with a single *bit* x_i and the entire network with a DTMC. With those basics we're ready to introduce some concepts that we want to study with a simulation using PRISM [KNP11].

We define a *network configuration* as a state of the DTMC with n processes, so a tuple $(x_n, \dots, x_1) \in \{0, 1\}^n$. In the ring may exist one or more processes that have *tokens* and those can be passed between processes p_i and p_{i+1} (so always to the right neighbor). A process p_i has a token if $x_i = x_{i-1}$ ($x_1 = x_n$ due to ring network structure). Finally, a network configuration is *stable* if $\exists! p_i$ that has a token.

From the above definitions follow these facts:

- given a network with n processes, the set of states S of the underlying DTMC has 2^n states. This can be proved saying that each state $s \in S$ has n components, each component has 2 possible value, hence $|S| = 2^n$;
- given a network with $n = 2k + 1$ processes, there no exists network configuration with $t = 2j$ tokens, with $j \leqslant k$. To see why we use a constructive proof:

  Proof. Let $(x_n, x_{n-1}, \dots, x_2, x_1)$ be a network configuration such that $n = 2k + 1$ and $(x_{n-1}, \dots, x_2, x_1)$ be a suffix with $t = 2j$ tokens, for some $j \in \mathbb{N}$ (which is perfectly legal, for instance $(0, 0, 1, 1, 0, 0)$, with tokens in p_6, p_4, p_2, p_1).

  Suppose $x_1 = 0$ (the same argument may be applied for $x_1 = 1$). We can have the following suffixes of even length:

  - $(0, x_{n-2}, \dots, x_2, 0)$, we can complete it by adding x_n such that:
    * $(0, 0, x_{n-2}, \dots, x_2, 0)$ in this configuration there are $2j + 1$ token (one more due to (x_n, x_{n-1}));
    * $(1, 0, x_{n-2}, \dots, x_2, 0)$ in this configuration there are $2j - 1$ token (one less due to breaking (x_1, x_{n-1}));
  - $(1, x_{n-2}, \dots, x_2, 0)$, we can complete it by adding x_n such that:
    * $(0, 1, x_{n-2}, \dots, x_2, 0)$ in this configuration there are $2j + 1$ token (one more due to (x_1, x_n));
    * $(1, 1, x_{n-2}, \dots, x_2, 0)$ in this configuration there are $2j + 1$ token (one more due to (x_n, x_{n-1})).

Adding x_n on configurations of even length we get $2j \pm 1$ tokens, which is an odd number. $\square$

1.2.2 *Protocol rule*

Let k be a generic execution step, the protocol consists of the following rules:

$$\begin{array}{ll} x_i^{(k+1)} := x_{i-1}^{(k)} & \text{if } x_i^{(k)} \neq x_{i-1}^{(k)} \\ x_i^{(k+1)} := \mathtt{random}(0,1) & \text{if } x_i^{(k)} = x_{i-1}^{(k)} \end{array}$$

where $x_i^{(k)}$ is the value of x_i at step k, $:=$ means assignment and $\mathtt{random}$ is a function that simulates an unbiased coin toss. In Table 1 we report a simple example of protocol execution applied to 5 processes, starting from the network configuration $(1,1,0,0,1)$.

step	(x_5,x_4,x_3,x_2,x_1)	Tokens owners	coin tosses
1	$(1,1,0,0,1)$	p_5,p_3,p_1	$x_1^{(2)} := 0, x_3^{(2)} := 1, x_5^{(2)} := 1$
2	$(1,0,1,1,0)$	p_3	$x_3^{(3)} := 0$
3	$(0,1,0,0,1)$	p_3	$x_3^{(4)} := 0$
4	$(1,0,0,1,0)$	p_4	$x_4^{(5)} := 0$
5	$(0,0,1,0,1)$	p_5	...

Table 1: Example of protocol execution

It is interesting to observe these facts:

- from network configurations $\forall i : x_i = 0$ and $\forall i : x_i = 1$ of length n, it is possible to reach every other state by applying the second rule of the protocol;
- the protocol doesn't stop when a stable network configuration is reached. In fact, in a stable configuration, let p_i be the token owner: depending its coin toss outcome, p_i can decide to pass the token to p_{i+1} or to keep it another turn (in the example reported above, if at step 4 would have been $x_4^{(5)} := 1$ the next configuration would have been $(0,1,1,0,1)$, with p_4 token owner again);
- in order to have an estimate number of steps necessary to reach a stable configuration we do a PRISM simulation, using the *Confidence Interval* method, with 1000 samples. Looking at the log:

```
Path length statistics: average 5.4, min 1, max 29
```

So the average number of necessary steps for reaching a stable configuration is 5.4.

1.2.3 *Interesting properties*

In this section we verify if Herman protocol satisfy some desired properties, in particular if a stable configuration is eventually reached or if it is reached within k steps.

Before get deep into properties verification it is important to understand that if we'd have used a *qualitative* model checker to verify if a stable configuration is *always* eventually reached, the result would be *reject*. To see why, let the DTMC be in the state $(1, 1, 1, 1, \dots, 1, 1)$ and, repeating the application of rule two of the protocol, for all p_i the coin toss results in 1, getting $(1, 1, 1, 1, \dots, 1, 1)$ again. There exists an infinite path on the DTMC which catch the previous behavior, hence that path is the counterexample for "from the initial state $(1, 1, 1, 1, \dots, 1, 1)$, is a stable configuration *always* eventually reached?"

One initial state $(1, 1, 1, \dots, 1, 1)$

In Table 2 we report the properties that we've verified with PRISM using a network with 7 processes, the label "stable" represent a predicate such that:

$$\begin{aligned}\text{stable is true} \leftrightarrow 1 =& (x_1 = x_2 ? 1 : 0) + (x_2 = x_3 ? 1 : 0) + (x_3 = x_4 ? 1 : 0) + (x_4 = x_5 ? 1 : 0) \\ &+ (x_5 = x_6 ? 1 : 0) + (x_6 = x_7 ? 1 : 0) + (x_7 = x_1 ? 1 : 0)\end{aligned}$$

Multiple initial states

In Table 3 we report the verification of the following property, which informally states: "what is the maximum/minimum expected time to reach a stable state, starting from any initial configuration in which there are k tokens?"

$$\mathtt{filter}(m, R =?[F"\mathtt{stable}"], "\text{k\_tokens}") \quad m \in \{max, min\}$$

that we've verified with PRISM considering all states of the DTMC as an initial state. The label "stable" is the predicate defined in the previous section and "k_tokens" represent a predicate $\mathtt{tokens}(k)$ such that:

$$\begin{aligned}\mathtt{tokens}(k) \text{ is true } \leftrightarrow k =& (x_1 = x_2 ? 1 : 0) + (x_2 = x_3 ? 1 : 0) + (x_3 = x_4 ? 1 : 0) + (x_4 = x_5 ? 1 : 0) \\ &+ (x_5 = x_6 ? 1 : 0) + (x_6 = x_7 ? 1 : 0) + (x_7 = x_1 ? 1 : 0)\end{aligned}$$

In Figure 1 we report a plot of the results shown in Table 3: when in the initial configuration are present 3 tokens then more steps are needed to reach a stable configuration. Observe that if the initial configuration contains exactly one token then no step is necessary (as reported in the first row of Table 3).

description	PRISM property	result
from the initial state, a stable state is reached with probability 1	$P_{>=1}[F"\mathtt{stable}"]$	true
from the initial state, a stable state is reached within 10 steps with probability 1	$P_{>=1}[F_{<=}10"\mathtt{stable}"]$	false
from the initial state, a stable state is reached within 10 steps with probability .5	$P_{>=}.5[F_{<=}10"\mathtt{stable}"]$	true
from the initial state, what is the probability to reach a stable state within 10 steps?	$P_{=?}[F_{<=}10"\mathtt{stable}"]$	.8757
what is the expected number of steps required for the self-stabilization algorithm to reach a stable state?	$R_{=?}[F"\mathtt{stable}"]$	5.4933

Table 2: Properties and their verification using a single initial state

1.3 DYNAMIC POWER MANAGEMENT

In this section we'll study a *dynamic power management* (DPM) system, a generalized version of a model presented in [QWP99] which concerns the model of a 3 state Fujitsu disk drive.

DPMs are interesting to study because they occur every time a device (or, more generally, a service provider) is present, such that the device has multiple power states with the following constraints:

- each state has a different power consumption associated to;
- each state has a different response time when a service is to be performed.

The analysis of those models has the goal to reach a reasonable trade off between the power consumption and the delivered level of service, that is, having power consumption as low as possible and quality of service as high as possible.

In the following sections we'll build three implementations, each of them with little differences respect the others, leaving some freedom degrees which allow us to understand how changes impact on power consumption and quality of service.

tokens	sat states (out of 128)	maximum over filter	minimum over filter
1	14	0	0
3	70	6.857	2.857
5	42	5.973	5.018
7	2	5.493	5.493

Table 3: Property verification about multi initial states

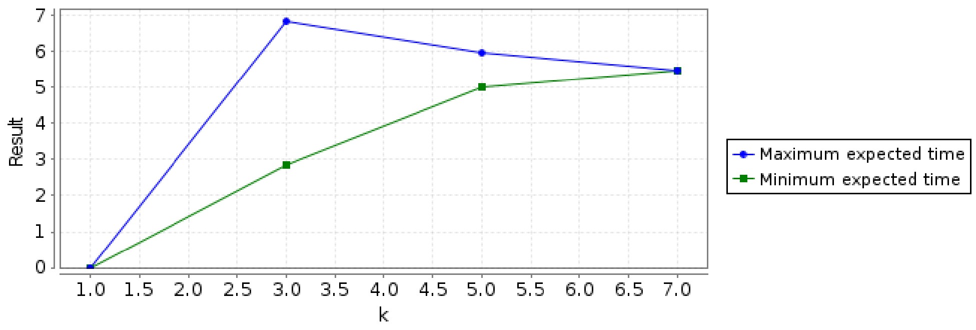

Figure 1: Max/min needed time to reach stable conf against tokens quantity

1.3.1 *First model: Service Queue and Service Provider*

The first model we've implement is composed by the following components:

- a *Service Queue* which registers generic service requests. We fix an upper bound $q_{max} = 20$ for the queue capacity and build a PRISM module with $0, \dots, q_{max}$ states, each of them represent the number of service requests asked up to a time t. The initial state of the *Service Queue* is 0, an empty queue, and a new request arrive every $.72$ seconds.
- a *Service Provider* which carries out requests. This component has three states: *sleep, idle, busy*, with *idle* is the initial state. We build a PRISM module with a variable such that it ranges over $\{0, 1, 2\}$ to cover the states of the component. The service provider serve a request every $.008$ seconds. The Service Provider makes a transition from *idle* to *busy* whenever a request arrives, while it makes a transition from *busy* to *idle* whenever it finishes to serve the last request.

This two components cannot take actions independently, they must synchronize for:

- receiving a request (in order to enque it correctly and moving the provider in a *busy* state because there's some work to get done);

- serving a request: this can result in a non empty queue if, in the previous state there are at least 2 requests in queue, therefore the provider has work to get done again; or it can result in an empty queue if, in the previous state there is exactly one request in queue, therefore the provider has no request to handle, so it can move to *idle* state.

Implications

From the modules' definitions reported above follow this facts:

- the total number of states of the PRISM system is $|\{0, \ldots, q_{max}\} \times \{0, 1, 2\}| = 63$ as confirmed by the log, looking for the dimension of the rate matrix;
- from the initial state, that is when the queue is *empty* and the Service Provider is *idle*, it is possible to reach only one state due to the synchronization on action *request*: the new state records that the queue contains exactly one element and the Service Provider is *busy*.

"Always accept requests and never serve them" path and property on sleep *state*

It is interesting to report a simple simulation case which study a "particular" path of the system, the one in which the system always choose to accept requests and never serves them. In Figure 2 we report this behavior. The plot is obtained exporting the path and loading the exported data in a R interpreter to plot the curve: the system, when the queue is full of requests, rejects all new requests.

To finish the discussion about this first model we study if it is possible for the service provider to enter in the *sleep* state. In order to do that we verify the following PRISM property, where $sp = 0$ is a true predicate if and only if the service provider is in *sleep* state:

$$P_{<=0}[F \quad sp = 0]$$

which holds, hence in this PRISM module the service locator never "*sleeps*".

1.3.2 *Second model: adding Power Manager*

Look at the previous model: the Service Provider decides for itself about its state transitions, using a strategy based on the number of requests present in the queue. In this new model we augment the previous behavior introducing a new component, a *Power Manager*, which interact with the Service Provider in order to drive its state transitions, especially between *sleep* and *idle* states.

We build a new PRISM model for the Power Manager, which synchronize the Service Provider to make the following transitions. Let $q_{trigger}$ be a freedom degree which represent a state of the queue:

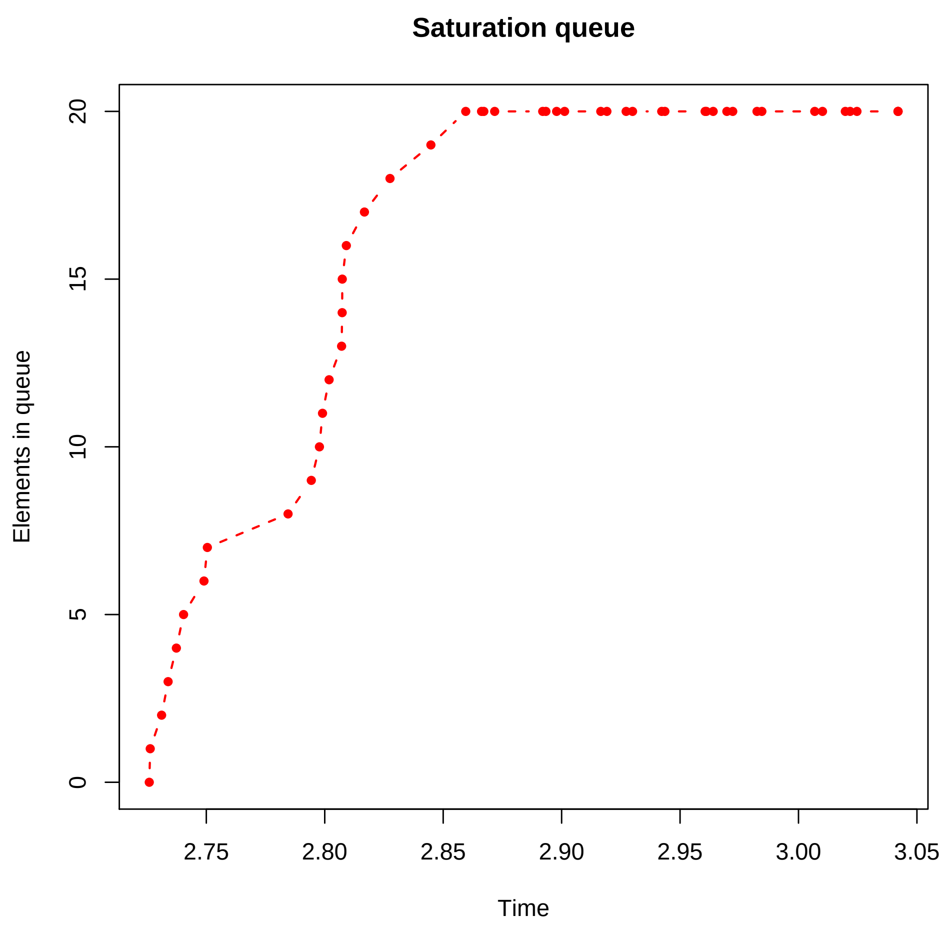

Figure 2: Queue saturation over time if a simulation path always choose to accept requests and never serves them

- *sleep* $\rightarrow$ $\mathtt{idle}$ if the current number q of waiting requests in queue is such that $q \geqslant q_{\mathtt{trigger}}$: this transition requires 1.6 seconds;
- $\mathtt{idle}$ $\rightarrow$ *sleep* if the current number q of waiting requests in queue is such that $q = 0$: this transition requires .67 seconds.

With the introduction of the Power Manager we check immediately that the Service Provider reach a *sleep* state sooner or later. We verify the following:

$$P_{>=1}[F \quad sp = 0]$$

which holds in this model while in first one doesn't.

In the following paragraphs we'll study some properties of interest which regard transient and long-run probability of: a full queue, expected queue size, cumulative missed requests and expected power (for the latter ones we need to introduce some *rewards structures*).

Transient and steady-state probability of a full queue

In Figure 3 we report a plot of an experiment concerning the *transient* probability of the queue being full in order to see if it stabilize after an amount of time. The experiment stress the following formula:

$$P_{=?}[F[T,T] \quad q = q_{max}]$$

The transient probability stabilizes around .001 as confirmed by the *green* curve which extends up to time 40, continuing the *blue* curve which stops at time 20 instead. It is possible to verify the previous result using another property which

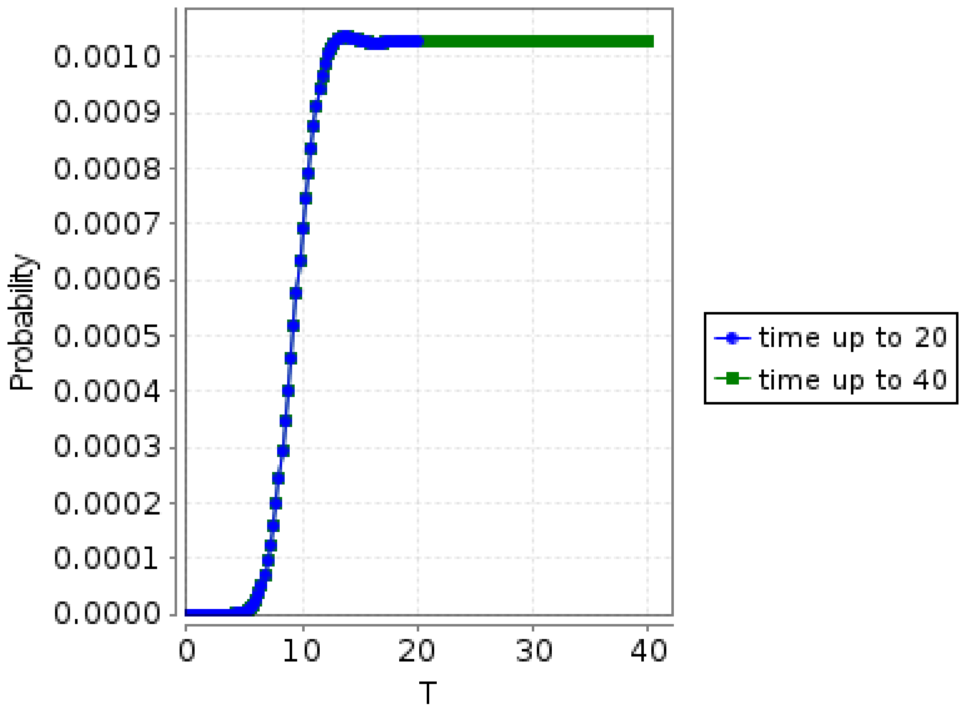

Figure 3: Transient probability of the queue being full

uses the *steady states* operator:

$$S_{=?}[q = q_{max}]$$

verifying the property we get .001028, in other words: "starting from the initial state, the steady-state (long-run) probability of having a full queue equals .001028". We reached this result using the Gauss-Seidel numerical method because the Jacobi method (the default one in PRISM) didn't converge in this case.

Transient and steady-state expected queue size

In Figure 4 we report a plot of an experiment concerning the transient expected queue size. To perform the experiment we need to introduce the following *rewards structure*:

```
rewards "queue_size"
true : q;
endrewards
```

which assigns as “rewards” to a state s the number q of requests present in queue when the system is in state s. After we stress the following formula:

$$R\{\text{"queue\_size"}\}_{=?}[I = T]$$

The transient probability stabilizes around 3.2 as confirmed by the *blue* curve which stops at time 20. It is possible to verify the previous result using another

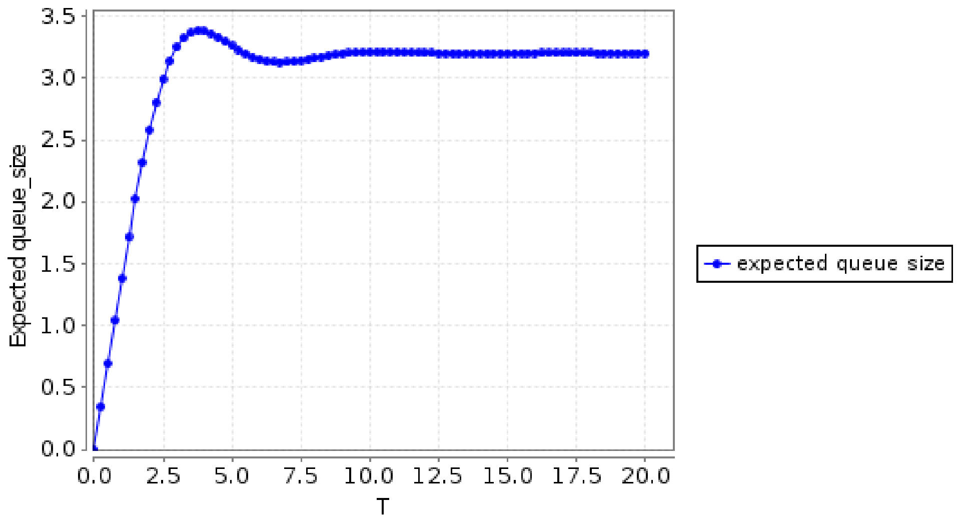

Figure 4: Expected queue size

property which uses the *steady states* operator:

$$R\{\text{"queue\_size"}\}_{=?}[S]$$

verifying the property we get 3.20388, in other words: “starting from the initial state, in the long-run the expected size of the queue equals 3.20388”. We reached this result using the Gauss-Seidel numerical method as did in the previous property.

Transient expected queue size with multiple values of $q_{trigger}$

The specification of this second model allow us to change $q_{trigger}$ in order to study how that change impact on the system. In Figure 5 we report the expected queue size (as done in the previous paragraph) where $q_{trigger} \in \{4, 6, 8, 10, 12, 14, 16, 18\}$. We see that greater $q_{trigger}$ implies more time is needed

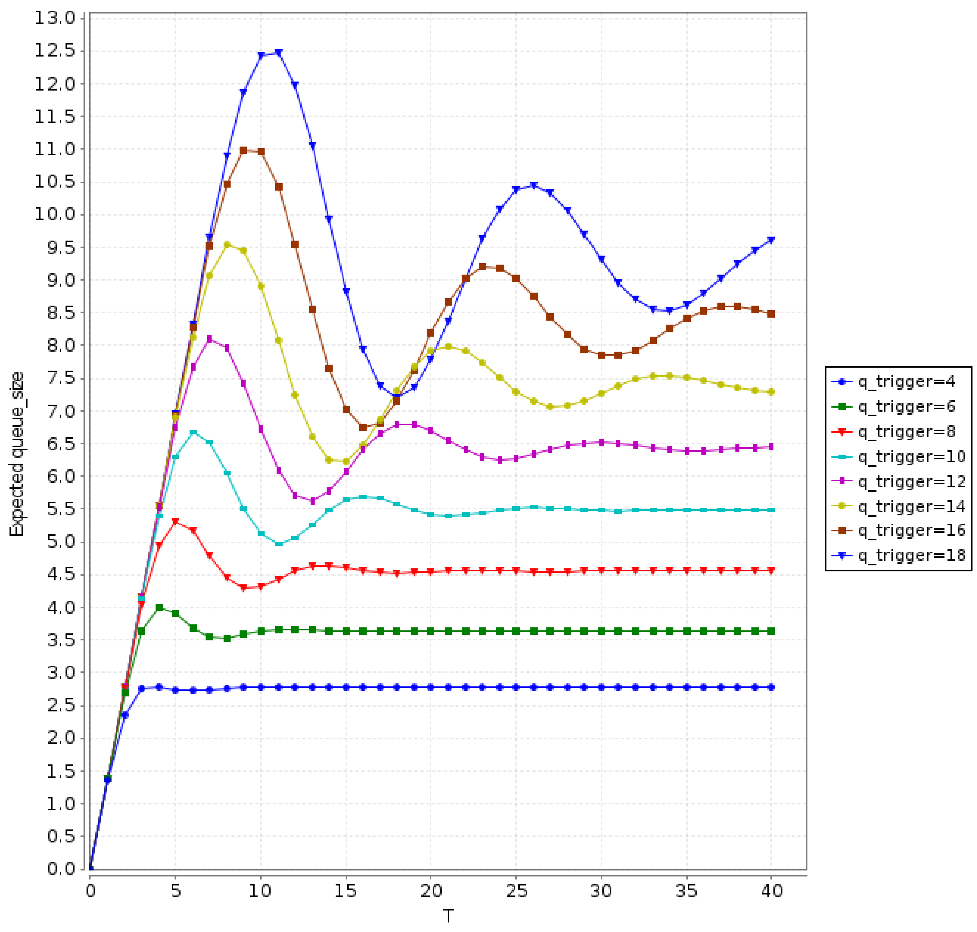

Figure 5: Expected queue size

to stabilize the queue size. Informally, the more the power manager waits before synchronizing with the Service Provider, the more "fluctuations" the system shows respect to queue size. Of those we can choose the curve for $q_{trigger} = 12$ because it has no common point with other curves (for instance the blue bigger one for $q_{trigger} = 18$ is better of the curve for $q_{trigger} = 16$ for $t \in [17, 22]$, but not for $t \in [0, \infty] \setminus [17, 22]$) and takes the queue size under half of the max dimension (in average).

Transient expected cumulative missed requests

In this paragraph we study another property about the expected cumulative number of lost requests due to a full queue. As done in the previous property, we use the same multiple values for $q_{trigger}$ and we introduce the following rewards structure:

```
rewards "lost"
[request] q=q_max : 1;
endrewards
```

which assigns a reward of 1 to a transition a such that:

- let $p, r \in S$ such that $p \xrightarrow{a} r$ and $p \vdash q = q_{max}$. Informally, the state from which the transition starts have to be a state where the queue is full;
- the transition happens due to a synchronization on the action *request*. The constraint on the action label is necessary because a request r is *lost* (or *missed*) if the queue is full when r arrives.

In Figure 6 we report the result of the experiment of the following property, which uses the C operator to get the cumulative value respect the reward structure *lost*:

$$R\{\text{"lost"}\}_{=?}[C <= T]$$

For greater values of $q_{trigger}$ the expected number of missed requests increase, in fact the more time the Power Manager waits before "weak up" the Service Locator, the more the queue gets bigger and more requests could be missed.

Transient expected cumulative power

In this paragraph we study the last property which concern the power consumption of the system. It isn't required to make any change to the model, all we have to do is to build a reward structure which catch the following specification (from [QWP99]):

> Energy is used at rate 0.13, 0.95 or 2.15 for Service Provider power states *sleep*, *idle* and *busy* respectively.
>
> Transitions between power states have a fixed energy cost: going from *sleep* to *idle* uses 7.0 units and from *idle* to *sleep* uses 0.067 units.

We definite the reward structure according the quote above:

```
rewards "power"
sp=0 : .13;
sp=1 : .95;
```

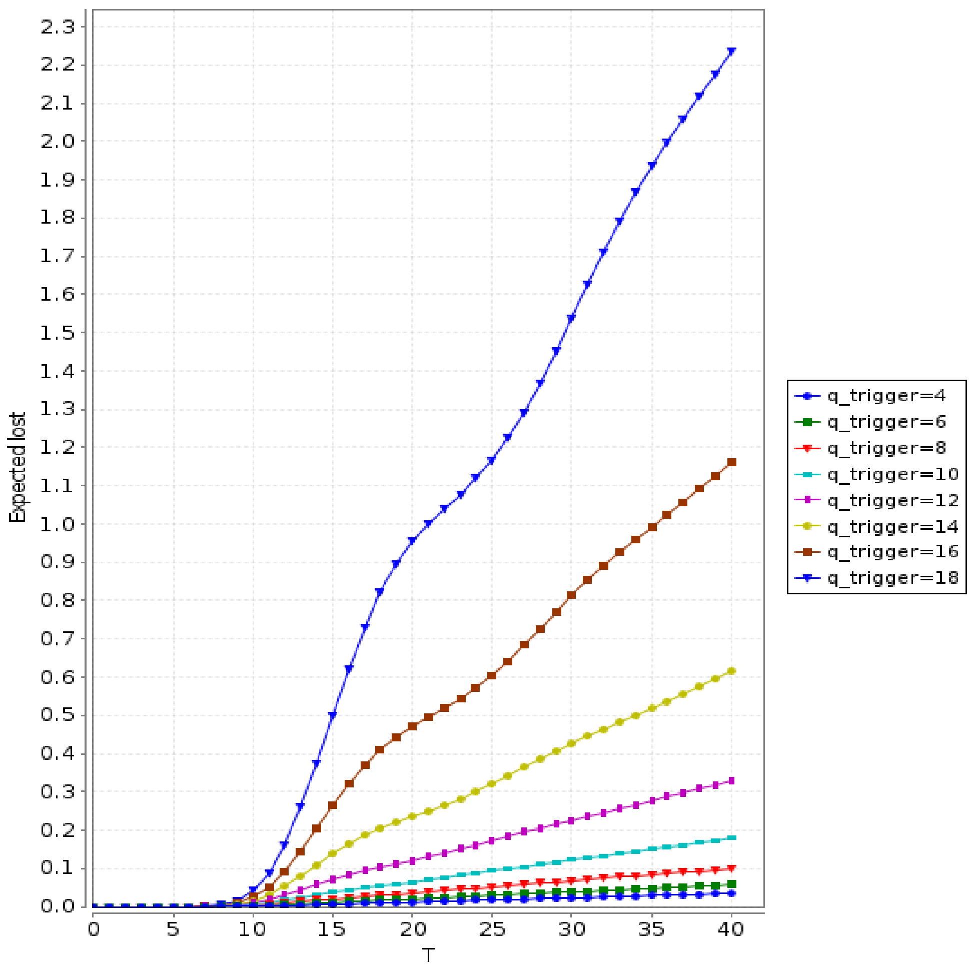

Figure 6: Expected cumulative missed number of requests

```
sp=2 : 2.15;
[sleep2idle] true : 7;
[idle2sleep] true : .067;
endrewards
```

In Figure 7 we report the experiment results of the following property which uses the structure just defined:

$$R\{\text{"power"}\}_{=?}[C <= T]$$

It is not surprising to see that the power consumption for $q_{trigger} = 18$ is the lowest: when $q_{trigger} = 18$ the expected queue size is very unstable, hence the queue would contain requests waiting for Service Provider. In that case the Service Provider will remain in its *busy* state, limiting the number of transitions between the other two states ($sleep \rightarrow idle$ is particular expensive) paying twice the cost of being in *idle* state.

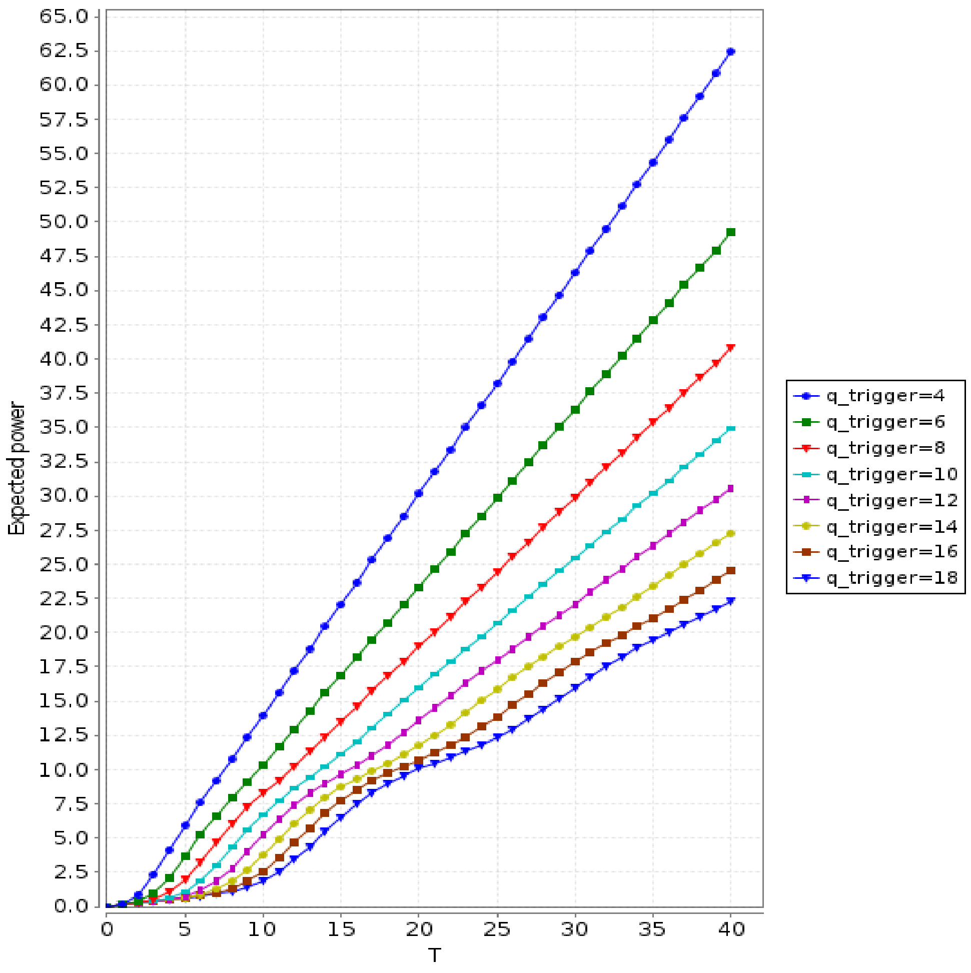

Figure 7: Expected cumulative power

1.3.3 *Third model: using* stochastic policies

In this paragraph we'll study the last model concerning DPM schemes. We use three components as in previous models but we introduce little changes in the Power Manager in order to apply *stochastic policies* on the transition `idle` $\rightarrow$ `sleep`, the more expensive transition. Specifically we build a new version of the Power Manager module such that:

- has a boolean variable which indicates if the transition has to be performed or not. This variable is used as a guard for the command which synchronize with the Service Provider to make it go to "sleep";
- the synchronization command to "wake up" the Service Provider now happen only when the queue is full, hence we remove the freedom degree $q_{\mathtt{trigger}}$ because it is useless in this new setting;

- has a new command that makes the Power Manager synchronize with the other components when the last request present in the queue is finally delivered. As update it use a new freedom degree $p_{\mathtt{sleep}} \in [0, 1]$ which represent the probability to set the new variable explained two points before, that is the probability for the transition $\mathtt{idle} \to \mathtt{sleep}$ to take place.

We can observe that for the minimum and maximum values of $p_{\mathtt{sleep}}$ we have, assuming the system is neither in its initial state (where the Service Provider is in *sleep* state) nor in the first interval time such that the queue hasn't reached the *full* state:

- if $p_{\mathtt{sleep}} = 0$ then the Service Provider will never reach the *sleep* state because the Power Manager always reset its internal variable that allow the $\mathtt{idle} \to \mathtt{sleep}$ transition;

- if $p_{\mathtt{sleep}} = 1$ then the Service Provider will reach the *sleep* state every time it can, because the Power Manager always set its internal variable that allow the $\mathtt{idle} \to \mathtt{sleep}$ transition.

We've repeated the same verification of the last three properties reported for the second model with $p_{\mathtt{sleep}} \in \{0, .2, .4, .6, .8, 1\}$. In Figure 8, Figure 9 and Figure 10 we report the plots which corresponds to those of the previous model about the expected queue size, the expected cumulative missed requests and the expected power consumption respectively.

We see that, for $p_{\mathtt{sleep}} = 0$ in the long run the expected queue size is 0 (because the Service Provider "never sleeps" and always does its job), the expected number of missed requests stabilizes around 2.25 (while for greater values of $p_{\mathtt{sleep}}$ the missed requests diverge) and the expected power is the greatest respect to greater values of $p_{\mathtt{sleep}}$ (being always active has its drawbacks).

Instead, for $p_{\mathtt{sleep}} = 1$ in the long run the expected queue size is the greatest (because the Service Provider "sleeps as much as it can"), the expected number of missed requests diverges and the expected power is the lowest (being "as lazy as possible" reduce power consumption).

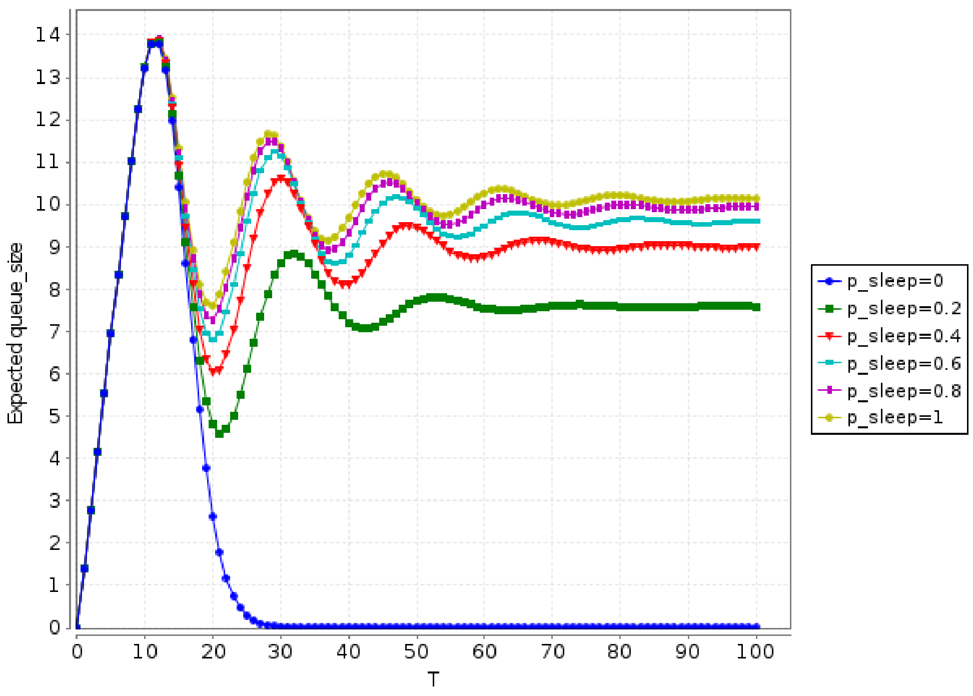

Figure 8: Expected queue size with multiple p_{sleep}

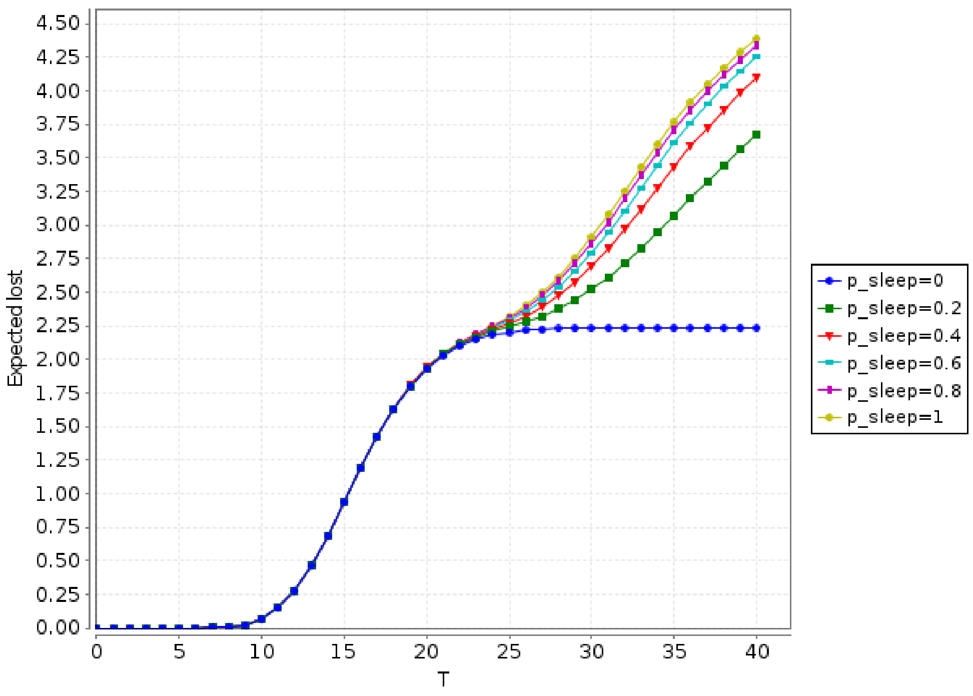

Figure 9: Expected missed requests with multiple p_{sleep}

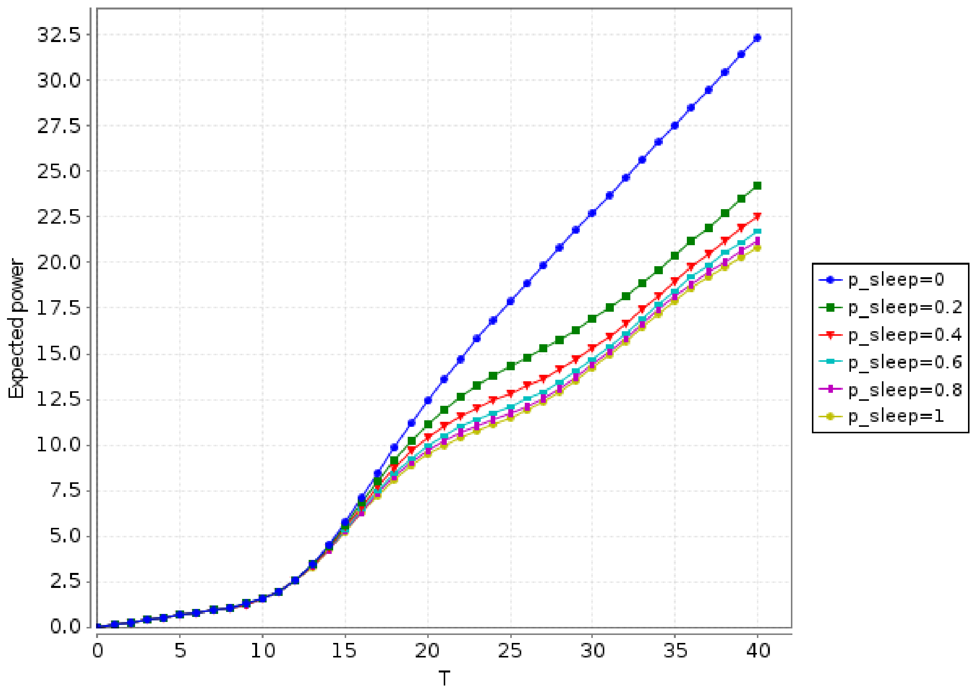

Figure 10: Expected power with multiple p_{sleep}

2 QUALITATIVE PART

In this part of the document we'll talk about the Hennessy-Milner logic over CCS processes and its extension with fix point operators. In a first step we review the foundamentals of the logic, giving the abstract syntax and an interpretation function. In a second step we talk about our implementation of the logic using the *OCaml* programming language and in a third step we apply our implementation to two simple case studies.

2.1 μ-calculus definition

Following [MOSS99], μ-calculus formulae are constructed according to the following grammar:

$$\phi := \mathtt{tt} \mid \mathtt{ff} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle a \rangle \phi \mid [a]\phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$

where $a \in Act$, $X \in Var$, $\mathtt{tt}$ means *true* and $\mathtt{ff}$ means *false*. The fix point operators $\mu X.\phi, \nu X.\phi$ bind free occurrences of variable $X \in Var$.

Formulae that can be composed using the syntax above are interpreted over *Labeled Transition Systems*. Given an *Labeled Transition Systems* $T = (S, Act, \rightarrow)$, we interpret a *closed* formula ϕ as the subset of states whose make ϕ true while we interpret an *open* formula using a function (also called *environment*) ρ such that $\rho : Var \rightarrow 2^S$. Roughly speaking, $\rho(X)$ interprets a free variable X contained in a formula ϕ by a subset of S, representing an assumption about the set of states satisfying the sub formula X.

Let $T = (S, Act, \rightarrow)$ be a *Labeled Transition Systems*, ϕ a formula and ρ an environment, we define the set $\mathcal{M}_T(\phi, \rho) \subseteq S$ satisfying ϕ inductively as follows:

$$\begin{aligned}
\mathcal{M}_T(\mathtt{tt}, \rho) &= S \\
\mathcal{M}_T(\mathtt{ff}, \rho) &= \emptyset \\
\mathcal{M}_T(\phi_1 \wedge \phi_2, \rho) &= \mathcal{M}_T(\phi_1, \rho) \cap \mathcal{M}_T(\phi_2, \rho) \\
\mathcal{M}_T(\phi_1 \vee \phi_2, \rho) &= \mathcal{M}_T(\phi_1, \rho) \cup \mathcal{M}_T(\phi_2, \rho) \\
\mathcal{M}_T([a]\phi, \rho) &= \{s \in S : \forall s' : s \xrightarrow{a} s' \rightarrow s' \in \mathcal{M}_T(\phi, \rho)\} \\
\mathcal{M}_T(\langle a \rangle \phi, \rho) &= \{s \in S : \exists s' : s \xrightarrow{a} s' \wedge s' \in \mathcal{M}_T(\phi, \rho)\} \\
\mathcal{M}_T(X, \rho) &= \rho(X) \\
\mathcal{M}_T(\mu X.\phi, \rho) &= \mathtt{fix}_\mu F_{\phi,\rho,X} \\
\mathcal{M}_T(\nu X.\phi, \rho) &= \mathtt{fix}_\nu F_{\phi,\rho,X}
\end{aligned}$$

where $\mathrm{fix}_\mu F_{\phi,\rho,X}, \mathrm{fix}_\nu F_{\phi,\rho,X}$ mean the least and greater fixed points of the function $F_{\phi,\rho,X} : 2^S \to 2^S$, which is defined as:

$$F_{\phi,\rho,X}(x) = \mathcal{M}_T(\phi, \rho'_{X,x,\rho})$$

where $\rho'_{X,x,\rho} : Var \to 2^S$ is defined as:

$$\rho'_{X,x,\rho}(X) = x$$
$$\rho'_{X,x,\rho}(Y) = \rho(Y) \quad \text{if } X \neq Y$$

Let $T = (S, Act, \to)$ be a finite-state *Labeled Transition Systems* (those for the set of states is finite), due to *Kleene fix point theorem* it is possible to characterize $\mathcal{M}_T(\mu X.\phi, \rho)$ and $\mathcal{M}_T(\nu X.\phi, \rho)$ by the following identities:

$$\mathcal{M}_T(\mu X.\phi, \rho) = \bigcup\{F^i_{\phi,\rho,X}(\emptyset) : i \geqslant 0\}$$
$$\mathcal{M}_T(\nu X.\phi, \rho) = \bigcap\{F^i_{\phi,\rho,X}(S) : i \geqslant 0\}$$

where $F^{i+1}_{\phi,\rho,X}(a) = F_{\phi,\rho,X}(F^i_{\phi,\rho,X}(a))$ and $F^0_{\phi,\rho,X}(a) = a$.

2.2 MODEL CHECKER IMPLEMENTATION

In this section we describe our implementation of a *local model checker* using the programming language OCaml. The development of the module has been structured in the following way:

1. built a new type Process with the necessary constructors to make *CCS processes*;
2. built a function unfold_process which allows to do one "unfolding step" respect to a free variable X bound by a Recur process constructor;
3. built a function next which consumes a Process and returns a list of possible computations the given process exhibits;
4. built a new type formula with the necessary constructors to make μ-calculus formulae;
5. built a function unfold_formula which allows to do one "unfolding step" respect to a free variable X bounded by a fix point operator;
6. built a function sat which consumes a process p and a formula f and produces the decision of $p \vdash f$, if p is a finite state process.

We would like to discuss briefly the details of sat function (the code can be found in subsection 2.3.3). It is the core of the model checker, using a local strategy explained in the Winskel's work [Win91]. It is a bit different respect the theory described above because a nice idea is introduced for fix point operators: Winskel allows a free variable to curry a set a states in a recursive formula specification, for example it is possible to write the formula $\nu X\{p_1, \dots, p_n\}A$, where $p_i \in S$ and A is a μ-calculus formula (where the X variable probably occurs into it). This is important due the following equivalence:

$$p \in \nu X.\gamma(X) \leftrightarrow p \in \gamma(\nu X(\{p\} \cup \gamma(X)))$$

where $\gamma : 2^S \to 2^S$ monotonic respect set inclusion relation. Roughly speaking, the equivalence says that a process p satisfies the recursively defined formula if and only if p satisfies a certain kind of unfolding of the recursively defined formula. Winskel asks to proof the following statement which is directly implemented in our implementation of sat function:

$$p \in \{p_1, \dots, p_n\} \to p \vdash \nu X\{p_1, \dots, p_n\}A$$

Proof. By contropositive, writing $\nu X\{p_1, \dots, p_n\}A$ as $\nu X(\{p_1, \dots, p_n\} \vee A)$ and $\cdot[\frac{T}{X}]$ is a function (written in post fix notation) such that $[\frac{T}{X}] : 2^S \to 2^S$, we have:

$$\begin{aligned}
& p \nvdash \nu X(\{p_1, \dots, p_n\} \vee A) \\
& \quad \updownarrow \text{ by def of relation } \vdash \\
& p \notin \bigcup \left\{ T \in 2^S : T \subseteq (\{p_1, \dots, p_n\} \cup A) \left[\frac{T}{X}\right] \right\} \\
& \quad \updownarrow \\
\nexists T \in (2^S \setminus \emptyset) : T \subseteq (\{p_1, \dots, p_n\} \cup A) \left[\frac{T}{X}\right] & \quad \text{such that} \quad p \in T \\
& \quad \updownarrow \{p_1, \dots, p_n\} \text{ cannot contain free occurrences of } X \in Var \\
\nexists T \in (2^S \setminus \emptyset) : T \subseteq \{p_1, \dots, p_n\} \cup A \left[\frac{T}{X}\right] & \quad \text{such that} \quad p \in T \\
& \quad \updownarrow \text{ in particular } T = \{p_1, \dots, p_n\} \text{ doesn't satisfy the condition} \\
& p \notin \{p_1, \dots, p_n\}
\end{aligned}$$

□

2.3 CASE STUDIES

In this section we report two models that we've checked using our implementation described in the section above.

2.3.1 *Non-bisimilar processes*

The processes under study are taken from [AILS07] and are reported in Figure 11. Those processes aren't bisimilar pairwise and, using the result that two processes are bisimilar if and only if they satisfy exactly the same formulae, we exhibit three witnesses of non-bisimilarity:

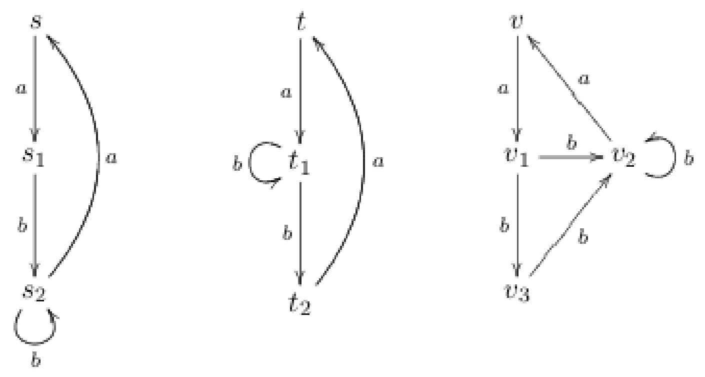

Figure 11: Three not bisimilar processes

```
Process s: RecX(a?.b?.RecY((b?.Y + a?.X)))
Process t: RecX(a?.RecY((b?.Y + b?.a?.X)))
Process v: RecX(a?.(b?.b?.RecY((b?.Y + a?.X)) + b?.RecY((b?.Y + a?.X))))

s isn't  bisimilar to t due to formula f= [a][b]<a>tt
s satisfy f: true
t satisfy f: false

s isn't  bisimilar to v due to formula f= [a]<b>[a]ff
s satisfy f: false
v satisfy f: true

t isn't  bisimilar to v due to formula f= [a]<b>[b]ff
t satisfy f: true
v satisfy f: false
```

2.3.2 *Car-train crossing*

In this model we catch the situation of a cross of a road and a rail. On both tracks a vehicle attempts to cross, a car and a train respectively. In order to cross, a vehicle have to synchronize with a "semaphore".

We build the necessary *CCS processes* in order to implement the model and we proceed to verify the following properties:

- a train eventually cross, written in μ-calculus using a min fix point:

$$\mu Z(\langle \mathrm{train\_cross}\rangle \mathrm{tt} \vee \langle \mathrm{Act}\rangle Z)$$

- it is no possible for the car and the train to cross simultaneously, written in μ-calculus using a max fix point:

  $$\nu Z(([\mathsf{train\_cross}]ff \vee [\mathsf{car\_cross}]ff) \wedge [\mathsf{Act}]Z)$$

- for each car c that attempts to cross, c eventually succeed, written in μ-calculus using a mixture of min and max fix points:

  $$\nu Z([\mathsf{car}](\mu Y(\langle \mathsf{Act}\rangle tt \wedge [-\mathsf{car\_cross}]Y)) \wedge [-\mathsf{car}]Z)$$

  It is interesting to note that the above formula does not use *alternating fix points*, that is the variable Z bounded by the out most ν does not appear in the context of the formula bounded by the innermost μ. This is not the worst case for the μ-calculus model checking (it is possible to write an equivalent *aCTL* formula for the above one using the "infinitely often" templates, while a formula with *alternating fix points* has not an *aCTL* equivalent).

Those properties are liveness, safety and fairness properties and involve the solution of a min fixed point, a max fixed point and a mixture of min and max fixed points respectively. We've written those properties using our OCaml formula constructors and verified them with the following results:

```
Process road: RecX(car?.up?.car_cross!.down!.X)
Process rail: RecY(train?.green?.train_cross!.red!.Y)
Process signal: RecZ((green!.red?.Z + up!.down?.Z))
Process crossing: (((RecX(car?.up?.car_cross!.down!.X) |
RecY(train?.green?.train_cross!.red!.Y)) | RecZ((green!.red?.Z +
up!.down?.Z))))\{green, red, up, down}
(((RecX(car?.up?.car_cross!.down!.X) |
RecY(train?.green?.train_cross!.red!.Y)) | RecZ((green!.red?.Z +
up!.down?.Z))))\{green, red, up, down}
Process crossing satisfy maxZ(([train_cross]ff or [car_cross]ff)
                                  and [Act]Z): true
Process crossing satisfy minZ(<train_cross>tt or <Act>Z): true
Process crossing satisfy maxZ([car](minY(<Act>tt and [-car_cross]Y))
                                  and  [-car]Z): true
```

2.3.3 sat *function code*

In this section we report the code of the sat function:

```
let rec sat p = function
| True -> true
| Not f -> not (sat p f)
| Or (f,g) -> (sat p f) || (sat p g)
| And (f,g) -> (sat p f) && (sat p g)
| Box (pred, f) ->
  let pred_matchers = List.find_all
    (fun (a, p') ->
match a with
| others -> pred a) (next p) in
  List.for_all (fun (a, p') -> sat p' f) pred_matchers
| Diamond (pred, f) ->
  let pred_matchers = List.find_all
    (fun (a, p') ->
match a with
| others -> pred a) (next p) in
  List.exists (fun (a, p') -> sat p' f) pred_matchers
| Min (var_name, seen_procs, f) ->
  if List.mem p seen_procs
  then false
  else sat p (unfold_formula var_name
                 (Min (var_name, (p::seen_procs), f)) f)
| Max (var_name, seen_procs, f) ->
  if List.mem p seen_procs
  then true
  else sat p (unfold_formula var_name
                 (Max (var_name, (p::seen_procs), f)) f)
| _ -> false;;
```

BIBLIOGRAPHY

[AILS07] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, NY, USA, 2007. (Cited on page 25.)

[Her90] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990. (Cited on page 6.)

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011. (Cited on page 6.)

[KNP12] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic verification of herman's self-stabilisation algorithm. *Formal Aspects of Computing*, 24(4):661–670, 2012. (Cited on page 5.)

[MOSS99] Markus Muller-Olm, David A Schmidt, and Bernhard Steffen. Model checking: a tutorial introduction. In *Proceedings of the Annual International Static Analysis Symposium*, volume 1694 of *lncs*, pages 330–354, September 1999. (Cited on page 23.)

[QWP99] Q. Qiu, Q. Wu, and M. Pedram. Stochastic modeling of a power-managed system: construction and optimization. In *Proc. International Symposium on Low Power Electronics and Design*, pages 194–199. ACM Press, 1999. (Cited on pages 9 and 16.)

[Win91] Glynn Winskel. A note on model checking the modal nu-calculus. *Theoretical Computer Science*, 83(1):157 – 167, 1991. (Cited on page 24.)