
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF COMPUTER SYSTEMS**
**ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ**


# STATISTICAL MODEL CHECKING OF APPROXIMATE COMPUTING SYSTEMS

**BACHELOR'S THESIS**
**BAKALÁŘSKÁ PRÁCE**


**AUTHOR** — **SERGIO PÉREZ HERNÁNDEZ**
**AUTOR PRÁCE**

**SUPERVISOR** — **Ing. JOSEF STRNADEL, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2019**

**Brno University of Technology**
Faculty of Information Technology

Department of Computer Systems (DCSY) Academic year 2018/2019

# Bachelor's Thesis Specification

21362

Student: **Pérez Sergio H.**
Programme: Shortterm study BSc.
Title: **Statistical Model Checking of Approximate Computing Systems**
Category: Modelling and Simulation
Assignment:

1. Summarize aspects and application areas of the so-called Statistical Model Checking (SMC) and approximate computing (AC) systems.
2. Identify SMC instruments suitable for modeling and analysis of AC systems.
3. Model representatives of a selected class of AC systems (such as approximate algorithms or circuits), check their properties by means of SMC and compare them with properties of "accurate" variants of such systems.
4. Evaluate your approach and discuss it from the applicability and validity viewpoints.

Recommended literature:

- According to the supervisor's recommendation.

Requirements for the first semester:

- Complete items 1 and 2 of the assignment, propose a model of a simple approximate system.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/
Supervisor: **Strnadel Josef, Ing., Ph.D.**
Head of Department: Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work: November 1, 2018
Submission deadline: May 15, 2019
Approval date: October 26, 2018

# Abstract

This thesis will be focused on the design, implementation and analysis of approximate computing system. This kind of systems allows some errors but improving performance and other aspects of it. UPPAAL SMC, a statistical model checking tool, will be used to implement and test the models.



# Abstrakt

This thesis will be focused on the design, implementation and analysis of approximate computing system. This kind of systems allows some errors but improving performance and other aspects of it. UPPAAL SMC, a statistical model checking tool, will be used to implement and test the models.



# Keywords

Approximate computing, statistical model checking, reliability, error rate, UPPAAL.



# Klíčová slova

Approximate computing, statistical model checking, reliability, error rate, UPPAAL.


# Reference

PÉREZ HERNÁNDEZ, Sergio. *Statistical Model Checking of Approximate Computing Systems*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Josef Strnadel, Ph.D.

# Statistical Model Checking of Approximate Computing Systems

## Declaration

I declare that I have prepared this Bachelor′s dissertation thesis independently, under the supervision of Ing. Josef Strnadel, provided me with further information. I listed all of the literary sources and publications that I have used.

.....
Sergio Pérez Hernández
May 10, 2019


## Acknowledgements

I would like to thank my supervisor Ing. Josef Strnadel for his effort and dedication in helping me throughout the process of creating my thesis.

# Contents

# Chapter 1

# Introduction

Since its beginning, information technology has been evolving to become in an essential field for the society of these days. Every moment, a huge amount of information is in movement, which has to be processed and stored. This could generate some problems because is possible that the right tools are not available for manage that volume, sometimes exorbitant, of data.
In this last decade, the 'Big Data' concept has been developing. It refers to the set of data o combinations of them which size, complexity and growing speed obstruct its capture, management, processing or analysis with conventional technologies and tools, such as relational databases and conventional statistics in the time that this data is useful.(7)
This is one of the reasons why it is necessary to use approximate computing. With it, we can sometimes admit a certain percentage of error in data processing so its size, complexity and speed can be adjusted to our possibilities. It is our task to determinate the accuracy that we can work with.
Approximate computing has been used in a variety of domains where the applications are error-tolerant, such as multimedia processing, machine learning, signal processing, scientific computing, etc.(13)

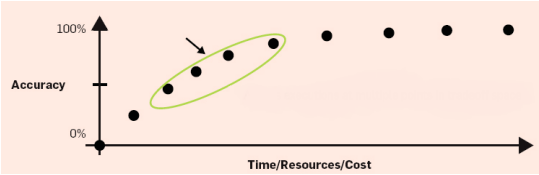



Figure 1.1: Approximate computing accuracy plot (4)

To observe the behaviour of some systems that use approximate computing, Statistical Model Checking method will be used. It does simulations of a system in a stochastic way, that is, non-deterministic, with certain conditions. Then, the results will be studied. Quantitative properties of stochastic systems are usually specified in logic that allow one to compare the measure of executions satisfying certain temporal properties with thresholds. The model checking problem for stochastic systems with respect to such logic is typically solved by a numerical approach that iteratively computes (or approximates) the exact measure of paths satisfying relevant subformulas.(5)

In this thesis, I will try to prove that approximate computing systems are useful is some specific fields. To do that, I will design some models of circuits that are used in computation and check their properties. The tool that will be used for the implementation will be UPPAAL, on version 4.1. It is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata.(9)

# Chapter 2

# Background

## 2.1 Statistical Model Checking

Model Checking is a recognized approach to guarantee the correctness of a system (6). It is based on algorithms that check whether all executions of a system satisfy some properties stated in specification logic. If this happens, the system is correct. Else, a bug is reported. Model checking can detect all bugs of a system, but it is generally slower.
Classical model checking techniques are Boolean, but this view is now obsolete. This has motivated the development of a series of new techniques, based in probability.

Statistical Model Checking has recently been proposed as an alternative to avoid an exhaustive exploration of the state-space of the model. The main idea is to conduct some simulations of the system, monitor them, and then use results from statistic area to decide if the system satisfies the property or not with some degree of confidence. SMC is a compromise between testing and classical model checking techniques. Furthermore, it is very simple to implement, it doesn't require extra modelling or specification effort and it allows to model check properties that cannot be expressed in classical temporal logics.

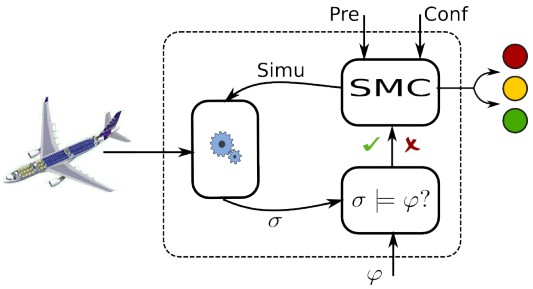



Figure 2.1: Scheme of Statistical Model Checking (6)

### Objective

A stochastic system $S$ and a property $\phi$ are considered (5). An execution of $S$ is a possibly infinite sequence of states of $S$. The objective is to solve the probabilistic model checking problem to decide if $S$ satisfies $\phi$ with a probability greater or equal to a certain threshold $\theta$.

The possible solutions of this problem depend on the nature of $S$ and $\phi$. There are three cases:

- $S$ is a white-box system, which means that one can generate as much executions of the system as we want. It is also assumed that $\phi$ doesn't contain probabilistic operators.
- $\phi$ can contain probabilistic operators.
- $S$ is a black-box system, which means that part of the probability distribution is unknown.

### White-box system

We assume that $S$ is a white-box system and $\phi$ is a bounded property. Let $B_i$ be a discrete random variable with Bernoulli distribution of parameter $p$. This variable can only have two different values: 0 and 1, with probabilities $P(B_i = 1) = p$ and $P(B_i = 0) = 1 - p$. Each variable $B_i$ is associated with one simulation of the system. The outcome for $B_i$, denoted $b_i$, is 1 if the simulation satisfies $\phi$ and 0 otherwise.

### SMC with $\phi$ containing probabilistic operators

There are another three possible extensions in this section:

- Unbounded case: It is based in *until property*,that requires that a property $\phi_1$ remains valid until a property $\phi_2$ has been seen. The problem is that the moment when $\phi_2$ will be satisfied is unknown. Hence, one must reason on infinite executions.
- Nested case: It is considered the problem of checking whether $S$ satisfied $\phi$ with a probability greater or equal to $\theta$, but now it is assumed that $\phi$ cannot be decided on a single execution. This means that $\phi$ cannot be model checked on a single execution, but rather depends on another test, so a way to nest tests is needed. To do it, Younes proposes the following theorem:
  *"Let $\psi = P_{\geq\theta}(\phi)$ be a property and assume that $\phi$ can be verified with Type-I error $\alpha$' and Type-II error $\beta$', then $\phi$ can be verified with Type-I error $\alpha$ and Type-II error $\beta$, assuming that the indifference region is of size at least:*

  $$((\theta + \delta)(1 - \alpha'), (1 - (1 - (\theta - \delta))(1 - \beta'))) \text{„}$$

  Hence, one has to find a compromise between the size of the indifference region of the inner test and the outer one. There are two facts about this:
  - The above result only works for systems that have the Markovian properties.
  - The complexity become exponential in the number of tests.
- Boolean combination: Only the conjunction and negation operations are considered. Younes also propose the two following theorems:
  - Conjunction theorem: *"Let $\phi$ be the conjunction of n properties $\phi_1$,...,$\phi_n$. Assume that each $\phi_i$ can be decided with Type-I error $\alpha_i$ and Type-II error $\beta_i$. Then, $\alpha$ can be decided with Type-I error $min_i(\alpha_i)$ and Type-II error $max_i(\beta_i)$"* With this idea, if we claim that the conjunction is not satisfied, this means that we have deduced that one operands is not. Furthermore, if we claim that the conjunction is satisfied, this means that it is concluded that all operands are satisfied.

– Negation theorem: *"To verify a formula $\neg\psi$ with Type-I error $\alpha$ and Type-II error $\beta$, it is sufficient to verify $\psi$ with Type-I error $\beta$ and Type-II error $\alpha$ „*

### Black-box system

This kind of systems are the ones whose probability distribution is not totally known and cannot be observed. It can be seen as a finite set of executions pre-computed and for which no information is available. Type errors indifference region cannot be used.
A solution to this problem is to conduct a SSP test assuming that the parameter n is fixed to the number of simulations that are given in advance. There are techniques to verify nested formulas over black-box systems. However, a technique for the verification of unbounded properties is still needed.

## 2.2 Approximate Computing

Approximate computing is a research agenda that seeks to better match the accuracy in system abstractions with the needs of approximate programs (8). The main challenge in approximate computing is forging abstractions that make imprecisions controlled and predictable without sacrificing its efficiency benefits. The objective is to design hardware and software around approximation.
The research in this area combines insights from hardware engineering, architecture, system design, programming languages, etc. Some of them are:

- Tolerance studies: This category shows how different parts of the application have different impacts on reliability and fidelity. Certain program components, especially those involve in control flow, need to be protected from all of approximation's effects.

- Exploit resilience in architecture: Hardware techniques for approximation can lead to gains in energy, performance, manufacturing yield or verification complexity. Hardware-based approximation strategies can be categorized according to the hardware component they affect: computational units, memories or the entire system architecture.

- Memory: Persistent Memories, where their storage cells can be worn out, approximate systems can reduce the number of bits they flip to lengthen the useful device lifetime. Also, memories like flash can use its probabilistic properties while hiding them from software. This memory approximation techniques typically work by exposing soft errors and other similar effects.

- Relaxed fault tolerance: Some circuit design techniques can be used to reduce the cost of redundancy by providing it selectively for certain instructions in a CPU, certain blocks in DSP or components of a GPU. Other use is to select critically information to allocate software-level error detection and correction resources.

- Microarchitecture: One set of techniques uses external monitoring to allow errors even in processor control logic. Other approaches compose separate processing units with different levels of reliability.

- Stochastic computing: It is an alternative computational model where values are represented using probabilities. A challenge in stochastic circuits is that reading and output value requires a number of bits that is exponential in the value's magnitude.

### Approximate systems

Most of work in approximate system architectures focuses on computation. Error tolerance in transient and persistent data is present in a broad range of application domains, from server software to mobile applications.
Memories have significant costs in performance, energy, area and complexity. It is because they need to ensure perfect data integrity 100% of the time.

Techniques that exploit data accuracy trade-offs are proposed to provide approximate storage and gain performance, energy and capacity:

1. Use multi-level cells in a way that enables higher density or better performance at the cost of occasional inaccurate data retrieval.

2. Use blocks with failed bits to store approximate data. To mitigate the effect of failed bits on overall value precision, the correction of higher-order bits is prioritized.

Approximate storage is applied to files and databases storage as well as transient data stored in main memory.

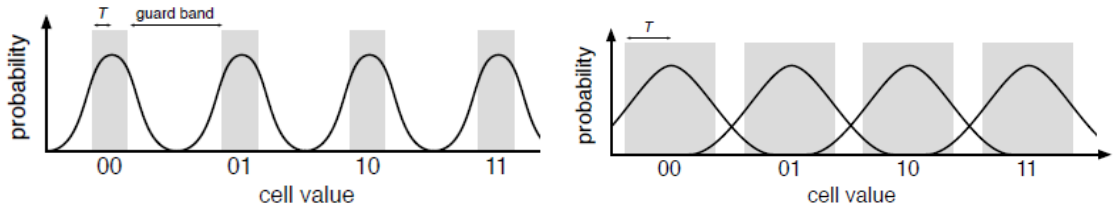
**Interfaces for approximate storage:** Modern non-volatile memory technologies exhibit properties that make them candidates for storing data approximately. By exploiting the synergy between these properties an application-level error tolerance, we can alleviate some of these technologies' limitations: limited device lifetime, low density and slow writes. When an application needs strict data fidelity, it uses traditional precise storage. Then, the memory guarantees a low error rate when recovering the data. When the application can tolerate occasional errors in some data, it uses the memory's approximate mode, in which data recovery errors may occur win non-negligible probability.
In approximate storage like Phase-change RAM and other solid-state, non-volatile memories, the application must determinate which data can tolerate errors and which data needs "perfect" fidelity.

**Approximate Main Memory:** Phase-change RAM and other fast, resistive storage technologies may be used as main memories. A wide variety of applications, from image processing to scientific computing, have large amounts of error-tolerant stack and heap data.

**Approximate persistent storage:** In this section, file systems, database management systems (DBMSs) or flat address spaces are considered. A data centre-scale image or video search database, for example, requires vast amounts of fast persistent storage. In occasional pixel errors are acceptable, approximate storage can reduce costs by increasing the capacity and lifetime of each storage module while improving performance and energy efficiency.

**Hardware interface and allocation:** The interface to approximate memory consists of read and write operations augmented with a precision flag. In the main-memory case, these operations are load and store instructions. In the persistent storage case, these are blockwise read and write requests. The memory interface specifies a granularity at which approximation is controlled. The compiler and allocator ensure that precise data is always stored in precise blocks.

**Approximate multi-level cells:** Phase-Change RAM and other solid-state memories work by storing an analog value and quantizing it to expose digital storage. In multi-level cell configurations, each cell stores multiple bits. For precise storage in MLC memory, there is a trade-off between access cost and density: many levels per cells requires more time and energy to access. Furthermore, protections against analog sources of error like drift can consume significant error correction overhead. But, where perfect storage fidelity is not required, performance and density can be improved beyond what is possible under strict precision constraints.




(a) Precise MLC (8)

(b) Approximate MLC (8)

This picture shows the range of analog values in a precise and approximate four-level cell. The shaded areas are target regions for writes to each level. The curves show the probability of reading a given analog value after writing one of the levels.

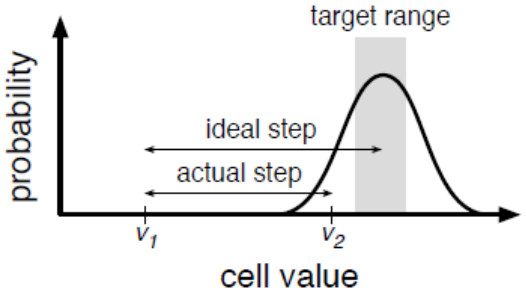



Figure 2.2: Single step in an iterative program-and-verify write (8)

In the figure above, the value starts at $v_1$ and takes a step. The curve presents the probability distribution from which the ending value, $v_2$, is drawn. Since $v_2$ lies outside the target range, another step must be taken.
An approximate MLC configuration relaxes the strict precision constraints on iterative MLC writes to improve their performance and energy efficiency.

# Chapter 3

# Selection of Implementation Areas and Means

## 3.1 Implementation Areas

My thesis will be divided into two parts. In the first one, I will try to evaluate the behaviour of an approximate logic multiplier, while in the second I will test the error rate of a DRAM memory depending on the refresh time of the electric current.

### 3.1.1 Approximate logical multiplier

A logical multiplier is a circuit that, from a series of inputs that represents a binary number, and another that represents another binary number, the multiplication of both numbers in the form of 0 and 1 is obtained. The length of the output will be the sum of the length of the two inputs. This, in case that the length of these numbers is very high, can lead to a high cost in the processing of the output. One solution can be the approximate logical multipliers.
The approximate logical multipliers are the same as the exact ones, except that they have fewer outputs. That means that we will have fewer costs but some multiplications won't be correct. This kind of multipliers are useful in systems that don't require a perfect computation, that is, they are tolerant of errors. Because of that, a certain amount of accuracy is sacrificed to reduce the area of the circuit and power consumption and increase the performance.(3)
To do the tests, an accurate multiplier and an approximate multiplier will be implemented. Both will have 4 inputs, where we want to multiply (B1, B0) with (A1, A0), obtaining 4 outputs: (Out3, Out2, Out1, Out0). This is the truth table for the exact logical multiplier:

| B1 | B0 | A1 | A0 | Out3 | Out2 | Out1 | Out0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

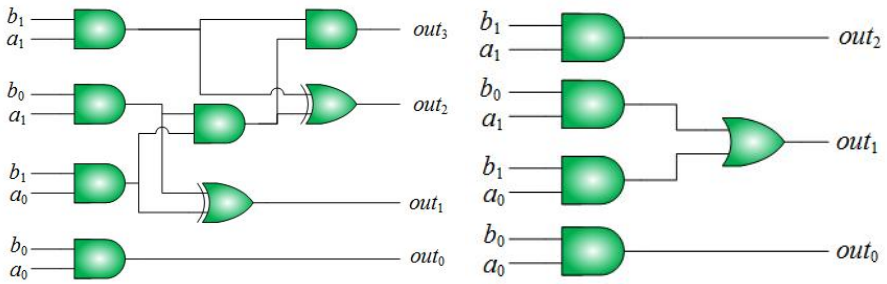
Table 3.1: Truth table of the accurate logical multiplier

We can see that, for Out3, all possible outputs are 0 except for the last one. What is done with the approximate multiplier is to delete this output. The last one will be transformed into (1,1,1), so, for these 16 possible outputs, only 1 will be incorrect:

| B1 | B0 | A1 | A0 | Out2 | Out1 | Out0 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 3.2: Truth table of the approximate logical multiplier

Then, from these tables, the logic circuits can be created:




(a) Accurate multiplier circuit (3) (b) Approximate multiplier circuit (3)

As can be seen in the pictures above, the accurate multiplier has 8 logic gates (6 AND gates and 2 XOR gates), while the approximate one only has 5 (4 AND and 1 OR), being this last one the simplest.

### 3.1.2 DRAM memory

A DRAM (*Dynamic Random Access Memory*) memory is a kind of RAM memory based on capacitors, which lose their charge progressively, so they need a refresh dynamic circuit that, every certain period, check this charge and replenish it. Its principal advantage is the possibility of build memories of a high density of positions and that work at a high speed. Like the rest of the types of RAM memories, it is volatile. It means that if the electrical power is interrupted, the stored information will be lost. The DRAM is widely used in digital electronics where low-cost and high-capacity memory is required. At the moment, it is one of the most used memories.(14; 15)
Each memory cell is the basic unit of each memory, able to store a bit in its logic circuits. Each cell has a transistor and a capacitor. Cells are organized in two-dimensional matrices, which are accessed through rows and columns.

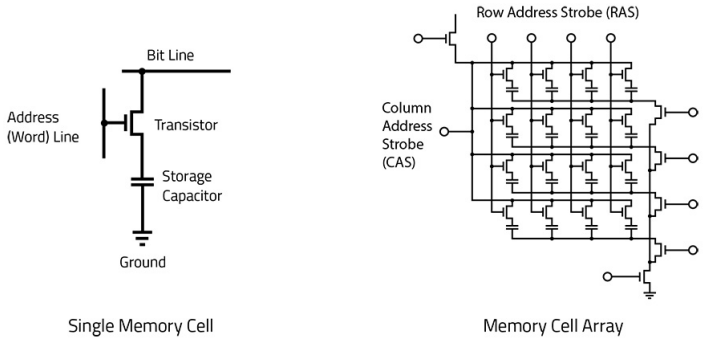



Figure 3.1: DRAM scheme (16)

If there is a charge inside the capacitor, it means that the logic value of the cell is 1. Otherwise, it is 0. The transistor connects or disconnects the capacitor. Over time, the capacitor will be discharging progressively. In the case that the voltage is below a threshold value, it will be assumed that the logic value of the cell is 0. That's why every so often it is necessary to recharge the capacitor. Therefore, the error rate of a DRAM memory will be checked depending on the time between the refresh periods.

## 3.2 Implementation Means

The implementation of this thesis will be in UPPAAL. It is a toolbox for validation and verification (via automatic model-checking) of real-time system (12).
The objective of this tool is to model a system using timed automata, simulate it and then verify properties on it. Timed automata are finite state machines with time (clocks). A system consists of a network of processes that are composed of locations. Transitions between these locations define how the system behaves. The simulation step consists of running the system interactively to check that it works as intended. Then, we can ask the verifier to check reachability properties.

This tool, at this moment, is in version 4.0, but there is also a version 4.1 that is in development, which includes an SMC extension. This last version is what will be used.

### 3.2.1 UPPAAL 4.0

UPPAAL is based on timed automata, that is a finite state machine with clocks. The clocks are the way to handle time. It is continuous and the clocks measure time progress. It is allowed to test the value of a clock or to reset it. Time will progress globally at the same pace for the whole system.

A system in UPPAAL is composed of concurrent processes, which are modelled as an automaton. This automaton has a set of locations. Transitions are used to change location. To control when to take a transition, it is possible to have a guard and a synchronization. A guard is a condition on the variables and the clocks saying when the transition is enabled. When a transition is taken, two actions are possible: assignment of variables or reset the clocks.

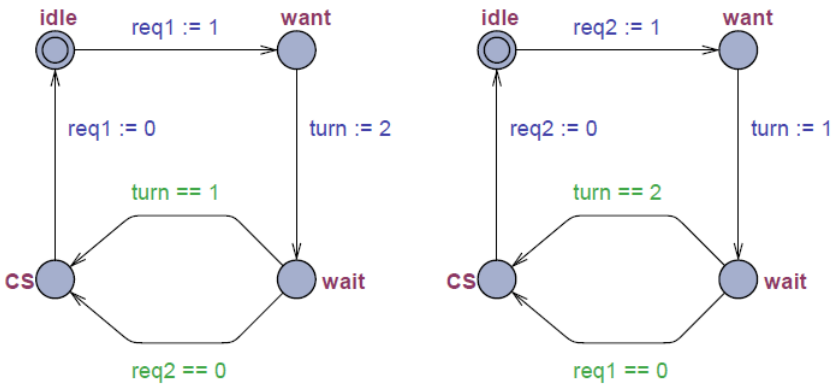



Figure 3.2: Example of a UPPAAL model (12)

#### Locations

There are different kinds of locations of UPPAAL (10):

- **Normal locations** (with or without invariants).
- **Urgent locations:** This kind of locations freeze time. It means that time is not allowed to pass. They are marked by a $U$ inside the circle.

- **Committed locations:** These locations also freeze time, but also, the next transition must involve an edge from one of the committed locations. They are useful for creating atomic sequences and for encoding synchronization between more than two components. They are marked by a *C* inside the circle.

There is also one (and only one) initial state. It is marked by a double circle.

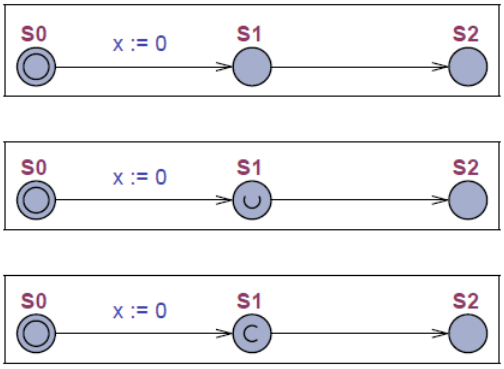



Figure 3.3: Example of a normal, urgent and committed states

### Verifying properties

To check if a property of our model is correct, UPPAAL has a verifier tool. The queries available in the verifier are (12):

- **E**<> ***p***: There exists a path where *p* eventually holds.
- **A**[ ] ***p***: For all path *p* always holds.
- **E**[ ] ***p***: There exists a path where p always holds.
- **A**<> ***p***: For all paths p will eventually holds.
- ***p*** $--> $ ***q***: Whenever p holds, q will eventually hold.

p and q are state formulas.
There is also a special query: **A**[ ] **not deadlock**, that checks for deadlocks (more transitions are not possible).

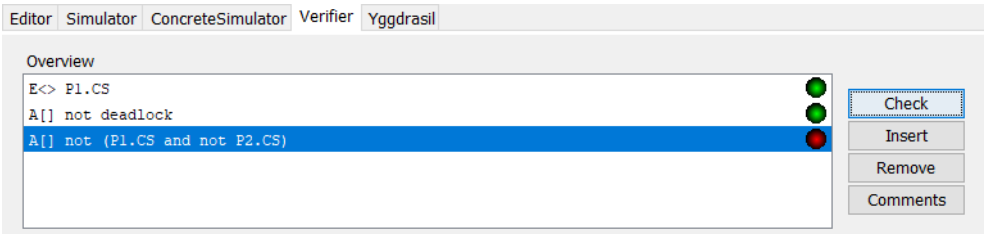



Figure 3.4: Verifier tool in UPPAAL

In this example we can see that the first and the second properties that we want to check are validated, but not the last one.

### 3.2.2 UPPAAL 4.1 (SMC)

The modelling formalism of UPPAAL SMC is based on a stochastic interpretation and extension of the timed automata formalism used in the classical model checking version of UPPAAL.(2) For individual timed automata components, the stochastic interpretation replaces the non-deterministic choices between multiple enabled transitions by probabilistic choices. Similarly, the non-deterministic choices of time delays are defined by probability distributions, which at the component level are given either uniform distributions in cases with time-bounded delays or exponential distributions in cases of unbounded delays.

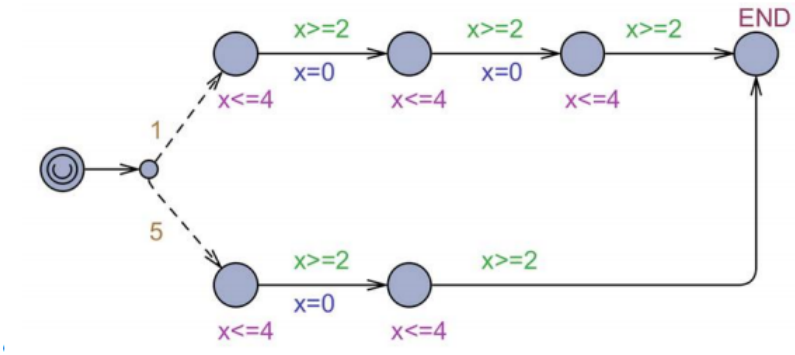



Figure 3.5: Example of a stochastic timed automata (2)

A model in UPPAAL SMC consists of a network of interacting stochastic timed automata. It is assumed that these components are input-enabled, deterministic and non-zero. These components communicate via broadcast channels and shared variables to generate networks of stochastic timed automata. The communication is restricted to broadcast synchronizations to keep a clean semantics of only non-blocked components which are racing against each other with their corresponding local distribution.

#### Additional verifying properties

**Simulation** In addition to the standard model checking queries, UPPAL SMC provides a number of new queries related to the stochastic interpretation of timed automata. In particular, it allows the user to visualize the values of expressions along simulated runs (evaluating to integers or clocks), providing insight to the user on the behaviour of the system so more interesting properties can be asked to the model-checker.
The concrete syntax applied in UPPAAL SMC is as follows:

**simulate** [<= *bound*]{$E_1$,...,$E_k$}

where $N$ is a natural number indicating the number of simulations to be performed, *bound* is the time bound on the simulations, and $E_1$,...,$E_k$ are the $k$ expressions that are to be monitorized and visualized.

**Probability estimation** The probability estimation algorithm of UPPAAL SMC computes the number of runs needed to produces an aproximation interval $[p - \varepsilon, p + \varepsilon]$ for $p = Pr(\Psi)$ with a confidence $1 - \alpha$. A frequentist interpretation of this result tells us that

if we repeat the interval estimation $N$ times, then the estimated confidence interval $(p \pm \varepsilon)$ contains the true probability at least $(1 - \alpha)$ $N$ in the long run $(N \to \infty)$
The syntax applied to check probability estimations is:

**Pr** $[bound](Expression)$

Also, UPPAAL gives you some plots, like the probability density distribution plot or the cumulative probability distribution plot.


Figure 3.6: Queries about probability estimation and simulation

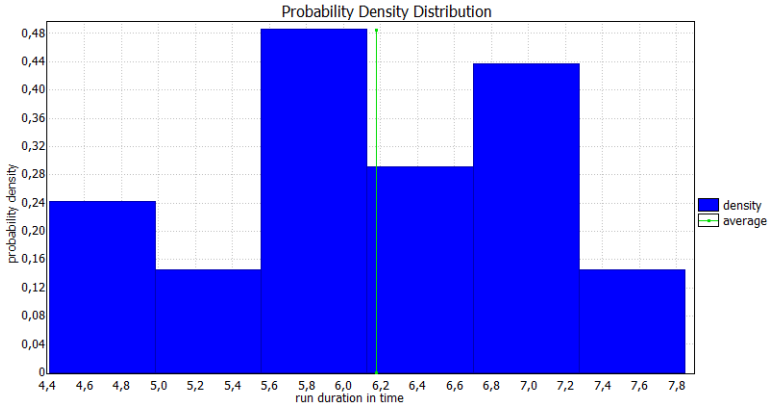



Figure 3.7: Probability density distribution plot

**Hypothesis testing** This approach reduces the qualitative question to test the null-hypothesis $H_0 : p \geq p_0$ against the alternative hypothesis $H_1 : p < p_0$. In UPPAAL SMC, we use the following query:

**Pr** $[bound](Expression) >= p_0$

**Probability comparison** The algorithm use the Wald test to compare probabilities, with the following query:

$\mathbf{Pr}[bound_1](Expression_1) = \mathbf{Pr}[bound_2](Expression_2)$

**Expected values** UPPAAL SMC also supports the evaluation of expected values of min or max of an expression that evaluates a clock or an integer value. The syntax is:

**E**[*bound;N*](**min**:*Expression*) or **E**[*bound;N*](**max**:*Expression*)

### 3.2.3 Other versions

#### UPPAAL TIGA

UPPAAL TIGA is an extension of UPPAAL and it implements the first efficient on-the-fly algorithm for solving games based on timed game automata with respect to reachability and safety properties. Though timed games for long have been known to be decidable there has until now been a lack of efficient and truly on-the-fly algorithms for their analysis (11). The algorithm is a symbolic extension of the on-the-fly algorithm suggested by Liu & Smolka for linear-time model-checking of finite-state systems. Being on-the-fly, the symbolic algorithm may terminate long before having explored the entire state-space. Also, the individual steps of the algorithm are carried out efficiently by the use of so-called zones as the underlying data structure. The tool implements various optimizations of the basic symbolic algorithm, as well as methods for obtaining time-optimal winning strategies (for reachability games).

#### UPPAAL Stratego

UPPAAL Stratego is a novel tool which facilitates the generation, optimization, comparison as well as consequence and performance exploration of strategies for stochastic priced timed games in a user-friendly manner. The tool allows for efficient and flexible "strategy-space" exploration before adaptation in a final implementation by maintaining strategies as first-class objects in the model-checking query language.(1)

# Chapter 4

# Proposed solutions

## 4.1 Approximate logical multiplier

Two models have been created for this implementation. The first one shows a more graphical way of how these logical systems work using transitions between different states. The second one uses transitions and functions.

### 4.1.1 Gate Network approach

With this implementation, the circuits shown in pictures 3.1a and 3.1b will be built. To do that, a model of each logic gate has been simulated. 4 random inputs will be generated randomly. Then, the outputs for the accuracy and approximate multiplier will be calculated by a series of comparisons and transitions using these gates. Finally, if these outputs are identical, the result of the approximate multiplier is correct. Otherwise, it is not. These will be checked the number of times the user wants, and, in the end, the number of successes will be count.

#### Global variables

- `count` (integer): Number of times that the system has calculated the outputs.
- `success` (integer): Number of times that the outputs of the approximate multiplier are the same as the three less significant output of the accurate multiplier.
- `input`(array of integer): Store the inputs that the system will use to calculate the outputs.
- `C0,C1,C2,D0,E0,E1` (integers): Auxiliary variables used in the process of the calculation of the outputs.
- `Out3ac,Out2ac,Out1ac,Out0ac` (integers): Outputs of the accurate multiplier.
- `Out2ap,Out1ap,Out0ap` (integers): Outputs of the approximate multiplier.
- Different broadcast channels used to send and receive signals between each gate.

### Templates

**Main** This template has one parameter: `loops`. It says the number of times that the system will calculate the possible outputs. Random inputs are selected using a normal distribution *N(0,1)*. First, 4 different random numbers are generated using this distribution and stored in a local array of integers `RandomV`. These numbers have the same probability of being positive or negative: *Pr(X<=0) = Pr(X>0) = 0.5*. So, if the number generated is negative, the number assigned for the corresponding position of `input` will be 0, and if it is positive, it will be 1. This is calculated by the function *setValues()*. Then, a signal will be sent to another template so it can start calculating the outputs for the accuracy and approximate multiplier. Finally, when these outputs are given back, they are compared by the function *check()*, except the most significant bit of the accurate model because it doesn't have any pair to compare with. If all are the same, one unit will be added to the variable `success`. Then, if the system has done its last iteration (seeing if `count` is equal to `loops`), it will finish. Else, the process will start again.
A clock `t` is added to the model, so an iteration will be done depending on the delay that it shows in the first state.

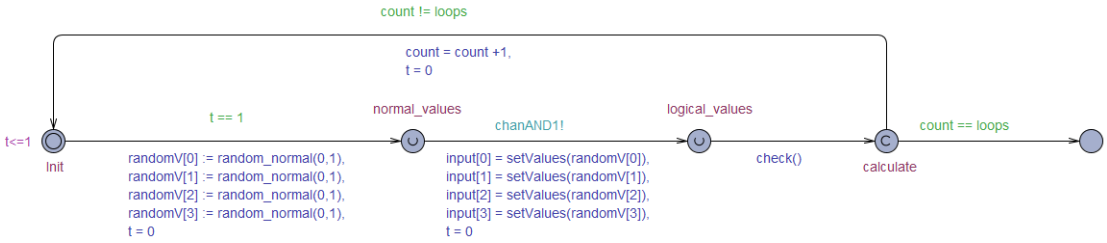



Figure 4.1: Main template

**AND** This template simulate the behaviour of an AND gate. It has 5 arguments:

- `chanInput` (broadcast channel): Channel that receives the signal so it can start working.
- `input1` and `input2` (integers): Inputs from which the output want to be got.
- `output` (integer): Output obtained from the transitions.
- `chanOutput` (broadcast channel): Send the signal so another template can start working.

In an AND gate, the only possibility that the output is 1 is that both inputs are also 1. So, when the signal of `chanInput` is received, two comparisons are established. The first one is that, if the value of the first input is 0, the output will be 0 and the signal for `chanOutput` will be activated. In case that it is 1, it is moved to an intermediate state where the second input intervenes. If it is also 1, then the output will be 1. Otherwise, it will be 0. Then, there is a transition to an auxiliary state `aux` and another one to the initial state, sending the signal `chanOutput` to the next template.

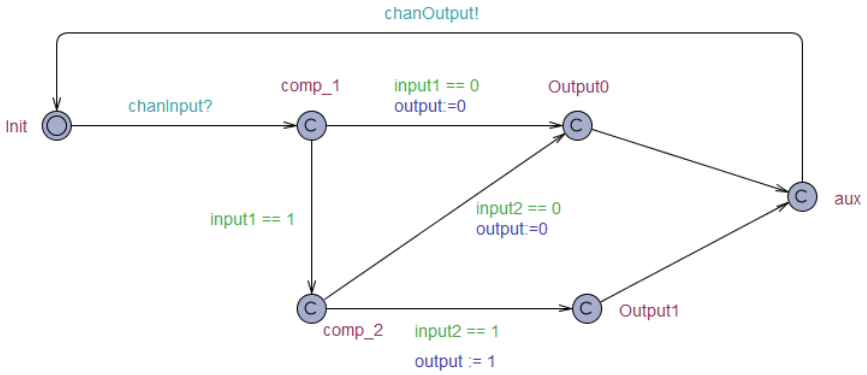



Figure 4.2: AND template

**XOR** Like the AND template, this shows how the XOR gate works. This kind of gate returns 1 if both inputs are different, and 0 if are the same. It has the same 5 arguments of the last template, but its functionality is different. Once `chanInput` receive the signed, the first comparison is done. if the first input is 0, we go to an intermediate state, and if is 1, we go to another intermediate state. For both states, if the second input is equal to the first one, then the output will be 0, otherwise, it will be 1. Then, a transition to an auxiliary state is done and another one to the initial state. The signal of `chanOutput` will be sent in this last transition.

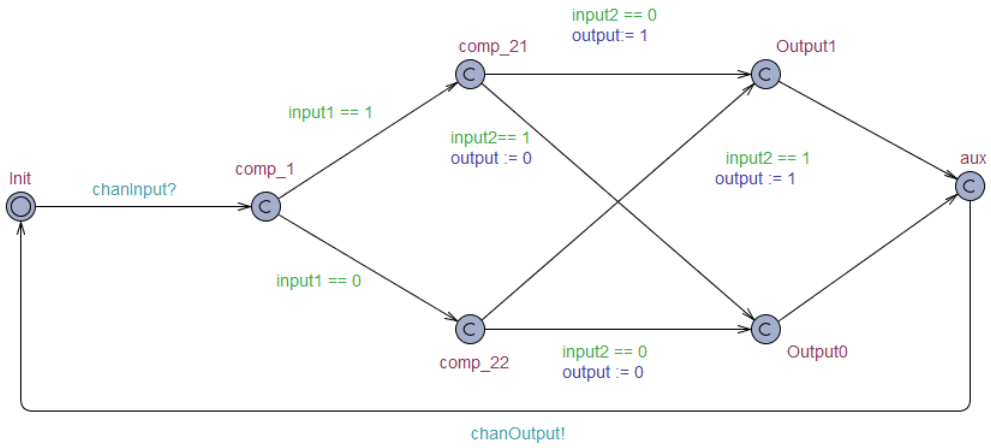



Figure 4.3: XOR template

**OR** This template has only 4 arguments because it is only used at the end of the simulation, so it doesn't have `chanOutput`. This kind of gates returns 1 if some of its inputs are 1, 0 in another case. Like the other templates, we start receiving the signal in `chanInput`. If the first output is 1, the value 1 is given to the output. If not, the second input is compared. If it is 1, the output will be 1, and if it is 0, the output will be 0. Then, as the other templates, a transition is done to an auxiliary state and going back to the first state.

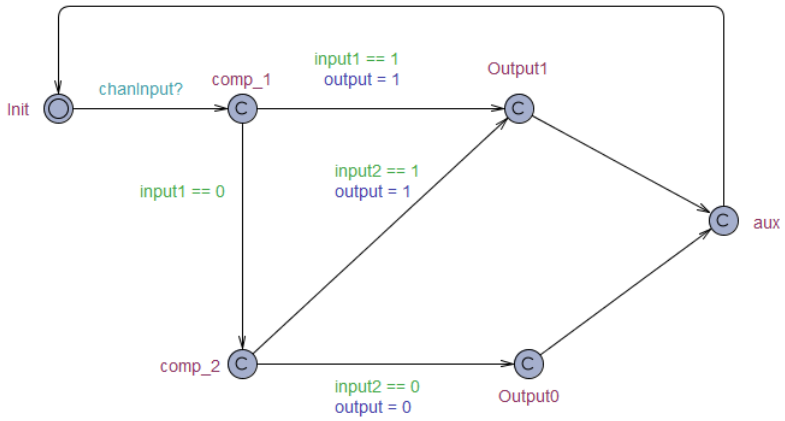



Figure 4.4: OR template

Finally, in the system declarations section, the different gates that encompass each circuit are declared. As it has been explained before, the accurate multiplier has 6 AND and 2 XOR gates, and the approximate multiplier has 4 AND and 1 OR gates.

```
//Accurate multiplier
and1ac = AND(chanAND1,A1,B1,C0,chanAND2);
and2ac = AND(chanAND2,A1,B0,C1,chanAND3);
and3ac = AND(chanAND3,A0,B1,C2,chanAND4);
and4ac = AND(chanAND4,A0,B0,Out0ac,chanAND5);
and5ac = AND(chanAND5,C1,C2,D0,chanXOR1);
xor1ac = XOR(chanXOR1,C1,C2,Out1ac,chanXOR2);
xor2ac = XOR(chanXOR2,C0,D0,Out2ac,chanAND6);
and6ac = AND(chanAND6,C0,D0,Out3ac,chanAND1ap);
```

Figure 4.5: Declarations of the accurate multiplier

```
//Approximate multiplier
and1ap = AND(chanAND1ap,A1,B1,Out2ap,chanAND2ap);
and2ap = AND(chanAND2ap,A1,B0,E0,chanAND3ap);
and3ap = AND(chanAND3ap,A0,B1,E1,chanAND4ap);
and4ap = AND(chanAND4ap,A0,B0,Out0ap,chanORap);
or1ap  = OR(chanORap,E0,E1,Out1ap);
```

Figure 4.6: Declarations of the approximate multiplier

### 4.1.2 Truth Table approach

This implementation is more focused on programming using functions, and no so much is transitions as the first implementation was. It is based in the search for the corresponding output in the truth tables (3.1 and 3.2) depending on the inputs selected. With it, you can check all possible values of the outputs and also analyze with which frequency the approximate multiplier fails.
In total, there are 4 templates. 2 of them are used to update the values of the outputs for the accurate and approximate multipliers respectively, the third establishes the inputs in a

sorted way so all the possible outputs can be seen clearly, and the fourth generate random numbers for the inputs.

### Global variables

- `update` (broadcast channel): Allows the system to generate the outputs once the inputs have been established.
- `bits` and `bits_aprox` (boolean arrays): They have length 8 and 7 respectively. Positions 0, 1, 2 and 3 of each array represents the inputs, and positions 4, 5, 6 and 7 (This last one only for the array `bits`) represents the obtained outputs.
- `bitsCovered` (Integer): Represents the number of possibilities that have been used when only the different possibilities are wanted to be seen.
- `equalBits` (Integer): Establish if the outputs of the approximate multiplier are the same as the obtained by the accurate multiplier.
- `errorRate` and `finalErrorRate` (doubles): The first one stores the error rate at every moment of the simulation, and the second one only at the end.
- The truth tables of both multipliers are defined as a two-dimensional array, with dimension $2^{\texttt{N\_INPUTS}} \times (\texttt{N\_INPUTS} + \texttt{N\_OUTPUTS})$, where `N_INPUTS` and `N_OUTPUTS` represents the number of inputs and outputs respectively.

```
bool tbl_mul2[TBL_PWR2[NIB_MUL2]][NIB_MUL2+NOB_MUL2] = {
/*    a    b          y    */
    {0,0, 0,0,    0,0,0,0},
    {0,0, 0,1,    0,0,0,0},
    {0,0, 1,0,    0,0,0,0},
    {0,0, 1,1,    0,0,0,0},
    {0,1, 0,0,    0,0,0,0},
    {0,1, 0,1,    0,0,0,1},
    {0,1, 1,0,    0,0,1,0},
    {0,1, 1,1,    0,0,1,1},

    {1,0, 0,0,    0,0,0,0},
    {1,0, 0,1,    0,0,1,0},
    {1,0, 1,0,    0,1,0,0},
    {1,0, 1,1,    0,1,1,0},
    {1,1, 0,0,    0,0,0,0},
    {1,1, 0,1,    0,0,1,1},
    {1,1, 1,0,    0,1,1,0},
    {1,1, 1,1,    1,0,0,1}
};
```

Figure 4.7: Truth table declaration of the accurate logical multiplier

### Templates

**Template set_inputs** This template has 5 parameters: `a0,a1,b0,b1`,integers, that represent the positions that the inputs will be stored in the array, and `dly`, integer, that represents the period of time used by a set of inputs and outputs.
The local variables are a clock `x`,`input`, integer, used to update the inputs, and a boolean

array `inCoverSet` used to check if the simulation should end.
The functions created for this template are:

- *update_bits()*: Update the values of `bits` depending if the actual value of `input` is divisible or not by a number (1 for the less significant input, 2 for the second, 4 for the third and 8 for the most significant one).
- *update_bits_aprox()*: Copy the values of the inputs of `bits` (First 4 registers) into `bits_aprox` so comparisons can be done.
- *update_input()*: Call the two functions mentioned before, establish the value TRUE to `inCoverSet[input]` and add 1 to `input`.
- *check()*: Check if the outputs of both multipliers are the same, and set a value to `equalBits` depending on it.
- *error()*: Calculates the error rate committed at a specific time of the simulation.
- *inCovered()*: Check if all possible states have been analyzed. This is done seeing if all the registers of `inCoverSet` have the value TRUE.

**Template set_inputs_random** It is similar to the previous one, but with some differences, because this template is used to generate $n$ random inputs. It has the same arguments of `set_input`, but with one more: `loops`, integer, that specifies the number of times that inputs will be generated.
As local variables, there are a clock `x`, an integer `count` that will be a counter and a boolean `stop` that will tell if the simulation has to finish.
There are the same functions of the first template, but changing some definitions. There will be commented only the ones which are different, omitting the others:

- *update_bits()*: It generates random inputs. It uses a normal distribution to do it, as in the first implementation. 4 random numbers are generated with a distribution N(0,1), that have the same probability of being positive or negative. In case that the number is negative, the input will be 0, otherwise will be 1.
- *inCovered()*: Check if the counter `count` is equal to the number of inputs that wanted to be generated, established by `loops`. If it is, then `stop` will be set with TRUE.
- *update_input()*: Calls *update_bits()* and *update_bits_aprox()*, increase the value of `count` 1 unit and calls *inCovered()*

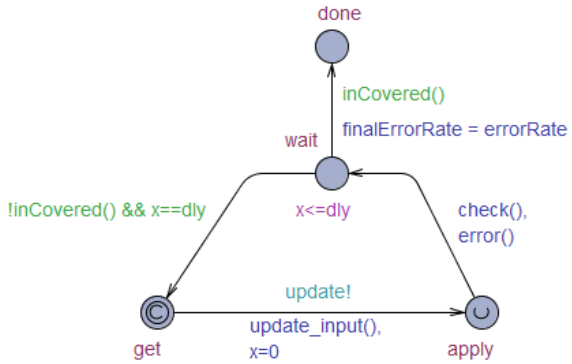



Figure 4.8: `set_inputs` and `set_inputs_random` template

Both templates have the same states and transitions, but, as it has been commented before, the functions they have are different.

**Template outputs_acc** This template has as parameters `a0,a1,b1,b0`, that represents the positions of the array `bits` where the inputs will be stored, `y0,y1,y2,y3`, that establish the positions of the same vector where the outputs will be, a two-dimensional boolean vector `ttbl` that references the truth table of the accurate multiplier, and an integer `dly` that set the delay between the different calculation of the outputs. As a unique local variable, there is a clock `x`.

`output_acc` only has one function, *bin2dec()*, that gets the binary values of the entries and returns a decimal number.

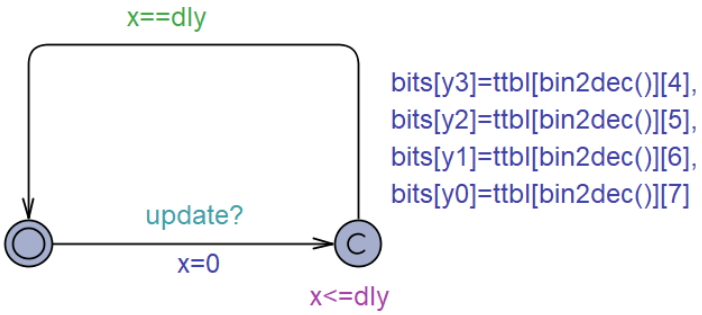



Figure 4.9: `outputs_acc` template

**Template outputs_aprox** This last template is similar to the last one. It has the same parameters, but in this case there is not `y3`, because only 3 outputs are obtained, `a0,a1,b0,b1,y0,y1` now set the positions of inputs and outputs in the vector `bits_aprox` and `ttbl` references the truth table of the approximate multiplier. It also has one local variable, the clock `x`, and the function *bin2dec()*.

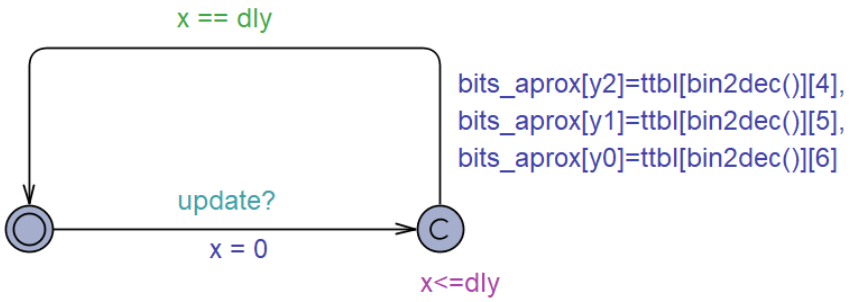



Figure 4.10: `outputs_aprox` template

### How it works

First, the simulation starts on the template `set_inputs` or `set_inputs_random`, depending on if it is wanted to obtain all possible outputs or $n$ random outputs. The initial state is *get*. A transition to the state *apply* is done. On it, the signal of the channel `update` is activated, which will allow the templates `outputs_acc` and `outputs_aprox` to do the first transition. Also, the function *update_inputs()* is called to update the inputs.
With the inputs updated, the templates `outputs_acc` and `outputs_aprox` will start working. In them, the outputs will be calculated, referencing the inputs to the respective truth table.
Once the outputs are obtained, there will be a transition to the state *wait*. In that transition, the system will check if the outputs of both multipliers are the same and it will calculate the error rate in that moment of the simulation.In *wait*, there will be checked if all possible combinations of inputs have been obtained (in the case of `set_inputs`) or if the number of desired outputs have been obtained (in the case of `set_inputs_random`). There are two possibilities:

- If the condition is not met, there will be a transition to the initial state when the delay time has been reached.
- If the condition is reached, a transition to the state *done*. `finalErrorRate` will be set with the current error rate and the simulation finish.

These are some possible results obtained with this implementation:

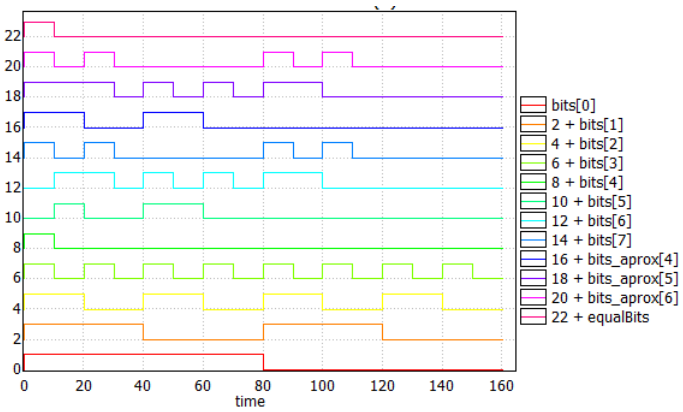



Figure 4.11: Simulation with `get_inputs`

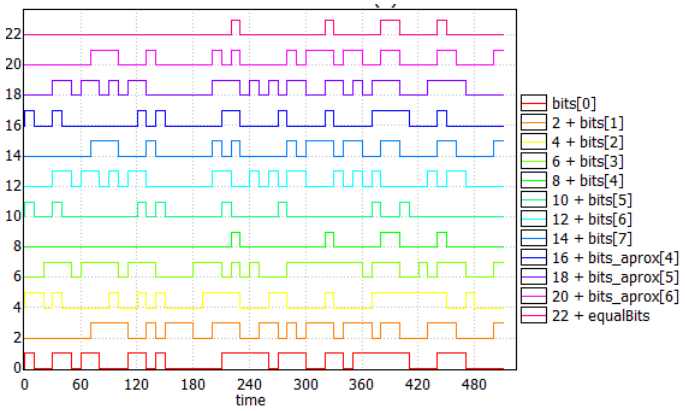



Figure 4.12: Simulation with `get_inputs_aprox`

In these implementations, more study cases could be added, such as generate inputs until the $x\%$ of outputs have been obtained, till an energy consumption level have been reached, etc.

## 4.2 DRAM memory

With this implementation, the functioning of a dynamic random access memory is wanted to be simulated, but on a small scale. Its objective is to show the different error rates that the cells have depending on the time of refresh that will be predetermined.

There are 4 templates. The first one only works as a switch, to start the simulation. The second is the one that simulates the behaviour of a cell memory, where the voltages and the different outputs are calculated. The third one represents the refresh circuit, used to give electrical power to the cells. The last one is used to do the tests, writing on the cells.

### Global variables

- `VCC_MAX` (double): Maximum voltage that a cell can have.
- `V_TRESH` (double): Threshold from which, if the voltage is higher than this value, the local value of the cell will be 1. Otherwise, it will be 0.
- `N_ROWS` and `N_COLS` (integers): Number of rows and columns that the memory will have.
- `tRow` and `tCol` (type definitions): Range of rows and columns, defined between 0 and `N_ROWS`-1/`N_COLS`-1.
- `uint8` (type definition): Set a range between 0 and $2^8 - 1$, so, [0,255].
- `tRefresh` (integer): Time that have to pass so the voltage of the memory fresh is refreshed.
- `row2Refresh` (`tRow`): Number of the row that have to be refreshed.

- `blockCnt` (integer in range [0,`N_COLS`]): Number of columns blocked. If it is 0, the bank is unlocked.
- `cVolt` (array of double[`tRow`][`tCol`]): Voltages of each memory cell.
- `cBit` (array of double[`tRow`][`tCol`]): Logical real value of each memory cell.
- `eBit` (array of double[`tRow`][`tCol`]): Logical expected value of each memory cell.
- `fail` (array of double[`tRow`][`tCol`]): Number of fails that have been committed in each memory cell.
- `loop` (array of double[`tRow`][`tCol`]): Number of times that have been checked if there is a fail.
- `errorRate` (array of double[`tRow`][`tCol`]): Error rate of each memory cell.
- `rowBuf` (array of boolean[`tCol`]): Row buffer, one per bank.
- `bit2rw` (array of boolean[`tCol`]): Bit that will be read of written in each column.
- `pwrUp` (broadcast channel): Activates the system.
- `rActivate` (array of bradcast channels [`tRow`]): Activates a row.
- `cOp` (broadcast channel): Signal to write, read or refresh over bank's buffer columns.

### Global functions

- *loadRBuf(*`tCol` *r)* : Activates bank's row *r* if the voltage of the column is higher than the threshold.
- *write8(*`uint8` *data)*: Writes *data* into the corresponding cells.

### Templates

**Template powerUp** This template doesn't have local variables or parameters. It just activates the signal `pwrUp` so the memory cells can start working.

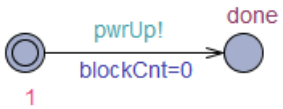



Figure 4.13: `powerUp` template

**Template mCell** `N_ROWS` × `N_COLS` memory cells are created with this template. It has as parameters `tRow` *r* and `tCol` *c*, representing the row and the column where the memory cell is. Like local variables, it has two clocks. `t0` counts the time till a writing or reading operation is done and `t1` counts the time till a memory cell is discharged.This template has

4 functions:

- *pwrUp_init()*: Set the voltage of the cell in 0.

- *targetV(bool r)*: It is used to charge the voltage of the cell. Returns the maximum voltage if an operation is been doing at that moment, otherwise returns 0.
- *updateLogicValue()*: Set the expected logical value of the cell.
- *calcError()*: Calculates the error rate of the cell.

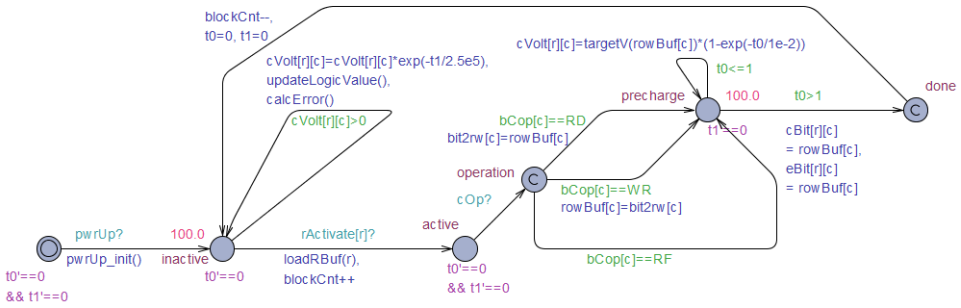



Figure 4.14: `mCell` template

**Template mRefresh** It doesn't have local variables or parameters. This template, when the time of refresh set by `tRefresh` has come, tells the memory cell that has to recharge its voltage.

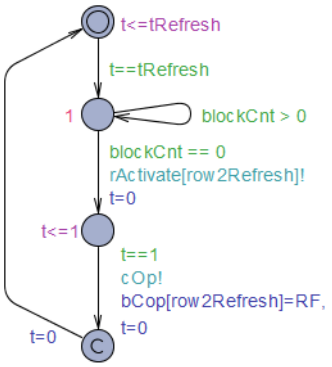



Figure 4.15: `mRefresh` template

**Template test** It is used to write values in different cells. Its aspect can change depending on the tests that wanted to be done. This is an example of writing in the second, third, and fourth cell of the first row.

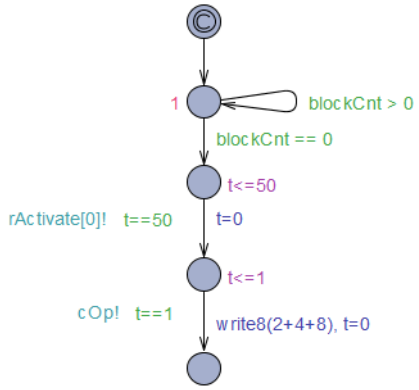



Figure 4.16: `test` template

### How it works

First, the cells that want to be written are chosen. To write on some cells, it is needed to wait that all the columns are unblocked. Then the row where it is wanted to write is activated and the signal of `cOp` is sent. The function *write8* is used, having as parameters the data that is wanted to be written separated by +, being 1 if the first cell is wanted to be written, 2 if it is the second, 4 if it is the third, ..., $2^{n-1}$ for the $n^{th}$ cell. Then, the `pwrUp` signal is sent in the `powerUp` template to the $N\_ROWS \times N\_COLS$ `mCell` templates. In this template, each cell is initialized with voltage 0 (`cVolt[r][c]` = 0), so it will wait in the state *inactive* till the row activation signal `rActivate[r]` is received. Then, the column is blocked and a transition is done to the state *active*. There, the cell has to wait again till the signal `cOp` to access to the state `operation`. Then, 3 options are possible, depending on the type of operation that will be done, specified in the array `bCop[c]`:

- `bCop[c]` = RD: Means that a reading will be done, so the content of `rowBuf[c]` will be copied to `bit2rw[c]`.
- `bCop[c]` = WR: The operation will be a writing. The content of `bit2rw[c]` will be copied to rowBuf[c].
- `bCop[c]` = RF: There will be a refresh. No further operations are needed.

In these 3 cases, there will be a transition to the state `precharge`. There, the cell is recharged with the maximum voltage set by `VCC_MAX`. Then, there will be a transition to `done`, where the real and expected logical value (`cBit[c]` and `eBit[c]`) will be set with the value of `rowBuf[c]`. Finally, the cell goes back to the state `inactive`, unblocking the column. In the next iteration, the voltage is higher than 0, so it will be discharging slowly, and also updating the expected logical value and calculating the error rate at that moment, till the moment that other operation will be done.
Lastly, the `mRefresh` template is used to refresh the model when is needed. The time of refresh is set by the variable `trefresh`. When that time comes, it is checked that all columns are unblocked. Then, the row that has to be refresh is activated and the operation RF is set in the array `bCop[row2Refresh]`. Finally, the `cOp` signal is sent to the `mCell` template to do the corresponding operation.
This is an example of the voltage, real logical value and expected logical value of a cell:

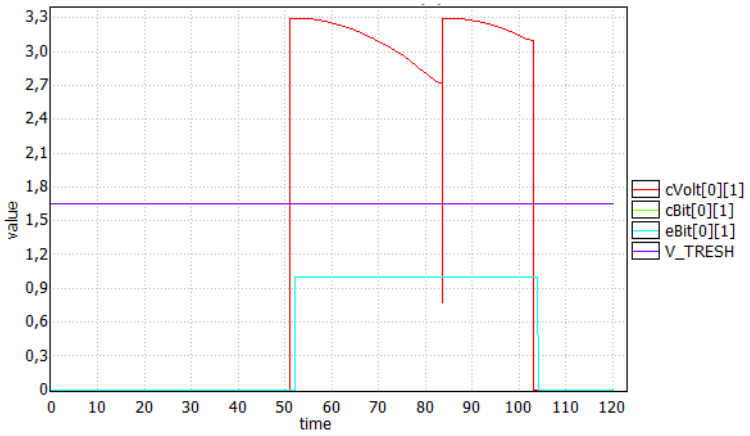



Figure 4.17: DRAM simulation example

In this plot, we can see that a writing is done when time is 50, and it has finished in 103 approximately. The time of refresh is set in 40. Both logical values, the real and the expected, are 1 when there is voltage in the cell. The voltage decrease with time, but when time is around 80, the voltage increase again. That is because 80 coincide with the time that the cell has to refresh.

# Chapter 5

# Evaluation

For the different tests that will be done on the models described before, the verifier tool of UPPAAL SMC will be used. With it, we can obtain various plots, confidence intervals, results of simulations using diagrams, etc.

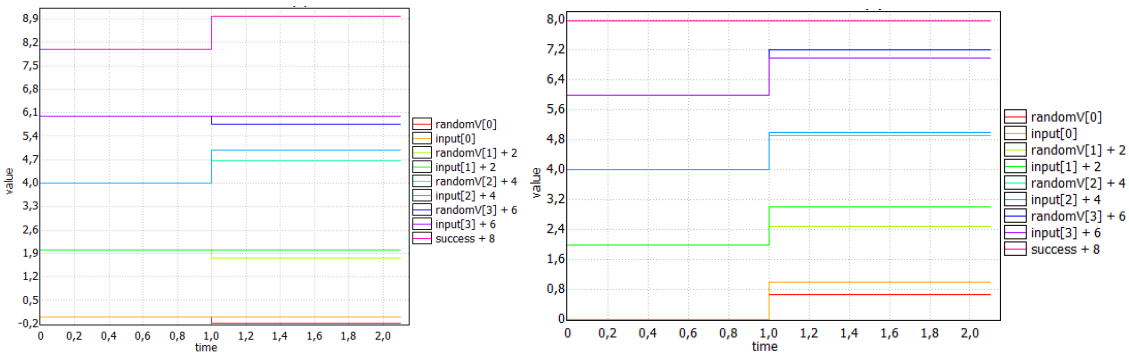
## 5.1 Tests on multiplier based on gate networks

### Verification of the model

First of all, it will be checked that the system behaves correctly. To do that, the parameter of the template `Main` will be set on 1, so only one iteration will be done. The following plot has been obtained with the command:

```
simulate[<=2] {randomV[0], input[0], randomV[1]+2,input[1]+2,randomV[2]+4,
              input[2]+4, randomV[3]+6, input[3]+6, success+8}.
```

On it, there will be represented, starting from below, the four random values obtained using a normal distribution. With them, if the value is positive, the input will be 1, and if it is negative, the input will stay in 0. At the top, the success variable is shown. These are two possible results:




(a) (b)

In figure 5.1a, we can see that, for the inputs 0, 1 and 3, the random values obtained are negative, and for the input 2 it has been positive, so the corresponding input is `0010`. With this combination of inputs, the approximate and the accurate multipliers have the

same outputs, so the variable `success` has increased one unit.
In the simulation that is represented in the second plot 5.1b, all random values have been positive, so the inputs of the system were `1111`. That is the only possible input which the outputs of the multipliers are different. That means that there has been a failure, so the variable `success` doesn't increase.

### Number of successes

In this test, I will see how many successes are obtained depending on the number of iterations that the system does. It is known that, theoretically, there is only 1 chance of 16 that the system fails (0.0625%). For the simulations, the next command has been used:

`simulate[<=`$N$`] {success}`

where $N$ represents the number of iterations done. These are the results:

| N | Successes | Fails | Error Rate |
|---|---|---|---|
| 10 | 10 | 10 | 0% |
| 50 | 46 | 4 | 0.08% |
| 100 | 94 | 6 | 0.06% |
| 200 | 183 | 17 | 0.085% |
| 300 | 278 | 22 | 0.073% |
| 400 | 376 | 24 | 0.06% |
| 500 | 471 | 29 | 0.058% |
| 750 | 710 | 40 | 0.053% |
| 1000 | 922 | 88 | 0.088% |
| 2000 | 1871 | 129 | 0.0645% |

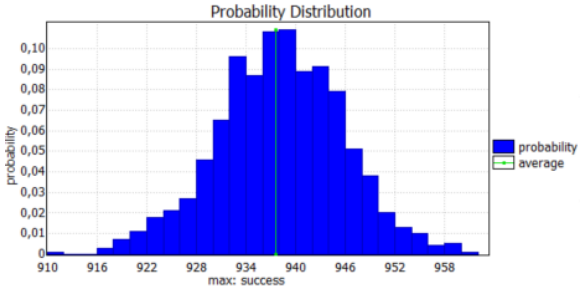
These results seem similar to the theoretical value, but it is needed to check it. To do it, I will do a hypothesis test. I will check the hypothesis $H_0 : \mu = 0.0625$ against $H_1 : \mu \neq 0.0625$. First, I calculate $t_{obs} = \frac{\overline{X}-\mu_0}{s}\sqrt{n}$, where $\overline{X}$ is the mean and $s$ is the standard deviation of the data. This statistic follows a distribution $t_{n-1}$.
For the data obtained, $\overline{X} = 0.06215$ and $s = 0.2496$. Then, $t_{obs} = \frac{0.06215-0.0625}{0.2496}\sqrt{10} = -0.00443$ follows a distribution $t_9$. Finally, I calculate the p-value, that is $2 * Pr(t_9 > -0.00443) = 0.9655931$. With a level of confidence of 5%, because of $0.9655931 > 0.05$, we can not reject $H_0$ and we can assume that $\mu = 0.0625$

### Behaviour of the number of successes

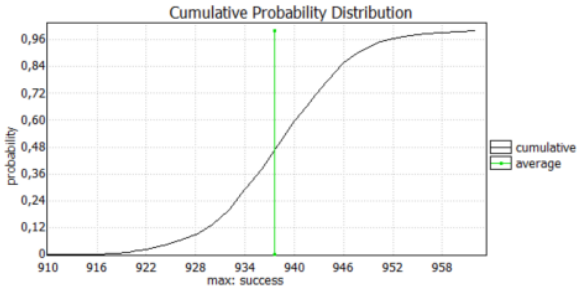
With this test, I will check what is the statistical behaviour of the number of successes. To do that, I calculate the expected values of the maximum of the variable `success` and then check the plots. I've used the next command in the verifier tool of UPPAAL:

`E[<=`$bound$`;`$N$`](max:success)`

where *bound* and $N$ are numbers big enough to have the significant results, and *bound* need to be at least as big as the parameter `loops` of the template `Main`. For the experiment, I've chosen 1000 for both numbers. These are some results:




(c) Probability Distribution Plot




(d) Cumulative Probability Distribution Plot

As numerical results, this simulation has been obtained as mean $\mu = 937.6$ and a confidence interval of 95% $[937.129, 938.071]$.

Observing the plots 5.1c and 5.1d, we can see clearly that the variable `success` follows a normal distribution with mean $\mu$, because of the shape of them, that seems to be the Gauss Bell in the first plot and an $S$ in the second.
Transforming now these values to error rate (doing the operation $1-(\frac{x}{1000})$), we obtain that the average error rate is 0.0624, and the confidence interval in 95% is $[0.061929, 0.062871]$. With this interval, we can see that the theoretical value is in it, and the mean is also really proximate to that number.

### Comparison of the resources used by both multipliers

In this test, I will compare the time needed by the system to calculate $N$ outputs of the accurate and the approximate multiplier separately. This time is given by UPPAAL at the end of the calculation. For that, I have modified the model, using only the definitions of the accurate or the approximate multiplier for each case. I've used the same command of the first test but varying the *bound*. These are the results:

| N | Accurate multiplier | Approximate multiplier |
|---|---|---|
| 100 | 0.032 | 0.016 |
| 1000 | 0.235 | 0.125 |
| 2000 | 0.36 | 0.281 |
| 3000 | 0.594 | 0.359 |
| 4000 | 0.75 | 0.563 |
| 5000 | 0.969 | 0.64 |
| 6000 | 1.093 | 0.75 |
| 7000 | 1.328 | 0.86 |
| 8000 | 1.485 | 1.015 |
| 9000 | 1.703 | 1.125 |
| 10000 | 1.969 | 1.344 |
| 11000 | 2.047 | 1.375 |
| 12000 | 2.266 | 1.5 |
| 15000 | 2.891 | 1.828 |

Table 5.1: Time used by the multipliers in seconds depending on the number of iterations

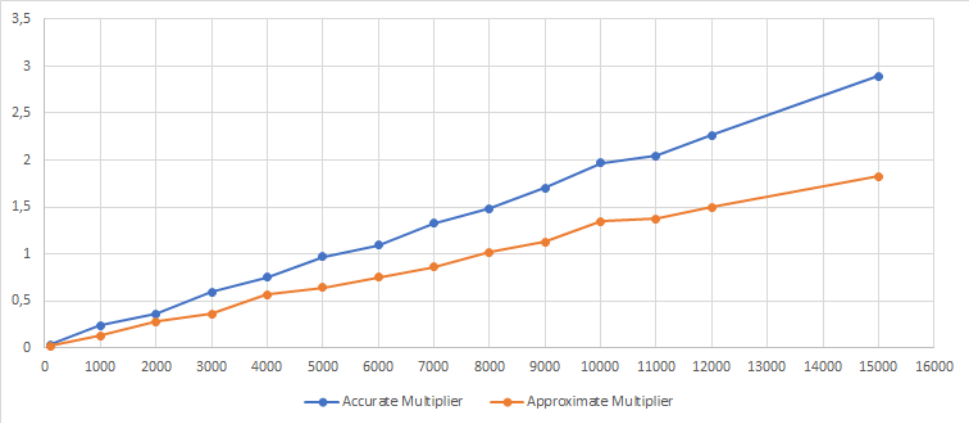



Figure 5.1: Plot of times of both multipliers

With this table and this plot, we can see that the time used by the accurate multiplier is higher than the used by the approximate multiplier in every point. To get more details, I will do a regression analysis of both multipliers. I will use R, a software for statistical computing, to do it. These are the commands used in that program:

```
N<-c(100,seq(1000,12000,1000),15000)
y1<-c(0.032,0.235,0.36,0.594,0.75,0.969,1.093,1.328,
      1.485,1.703,1.969,2.047,2.266,2.891)
y2<-c(0.016,0.125,0.281,0.359,0.563,0.64,0.75,0.86,
      1.015,1.125,1.344,1.375,1.5,1.828)
reg1<-lm(y1~N)
reg2<-lm(y2~N)
summary(reg1)
summary(reg2)
```

`reg1` represents the regression analysis for the accurate multiplier and `reg2` for the approximate multiplier. The outputs obtained are:

```
Call:
lm(formula = y1 ~ N)

Residuals:
      Min        1Q    Median        3Q       Max
-0.049512 -0.021371 -0.006535  0.019640  0.067443

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 3.946e-03  1.777e-02   0.222    0.828
N           1.898e-04  2.248e-06  84.428   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.03595 on 12 degrees of freedom
Multiple R-squared:  0.9983,    Adjusted R-squared:  0.9982
F-statistic:  7128 on 1 and 12 DF,  p-value: < 2.2e-16
```

Figure 5.2: Regression analysis results for the accurate multiplier

```
Call:
lm(formula = y2 ~ N)

Residuals:
      Min        1Q    Median        3Q       Max
-0.045408 -0.017707 -0.005499  0.005601  0.088501

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.968e-02  1.743e-02   1.129    0.281
N           1.236e-04  2.205e-06  56.041 6.87e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.03528 on 12 degrees of freedom
Multiple R-squared:  0.9962,    Adjusted R-squared:  0.9959
F-statistic:  3141 on 1 and 12 DF,  p-value: 6.872e-16
```

Figure 5.3: Regression analysis results for the approximate multiplier

In the first table, we can see that the results are significant, because $R^2$ is 0.9983 is close to 1 and the p-value of the model is low. The regression line obtained is $y1 = 0.003946 + 0.0001898N$, but observing the p-value of the intercept we can see that is so high (using a confidence level $\alpha = 0.05$). That is referred to the hypothesis of significant of that value. Interpreting it, we can consider that the intercept is 0, so the final regression line would be $y1 = 0.0001898N$.

In the second table, the $R^2$ is also close to 1, with a value of 0.9962, and the p-value of the model is close to 0, so the model is also significant. The regression line is defined by $y2 = 0.01968 + 0.0001236N$, but, the p-value of the intercept is also a big value, higher than $\alpha$, so it will be considered as 0. With this information, the final regression line is $y2 = 0.0001236N$.

Comparing both regression lines, the independent value of the second one increases slower than the independent value of the first as long as $N$ is growing. That means that **the approximate multiplier is more efficient than the accurate multiplier in terms of time**.

## 5.2 Tests on multiplier based on truth tables

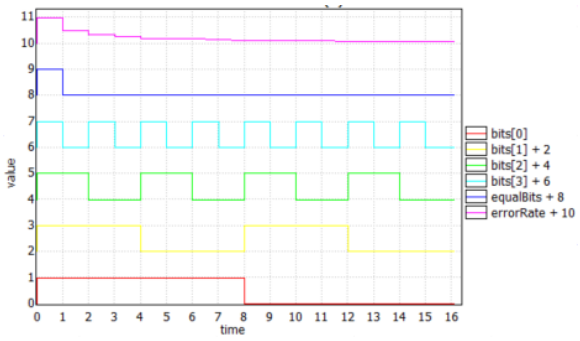
Tests on this model will be similar to the ones done in the previous model, so we can check that the results are correct.

### Verification of the model

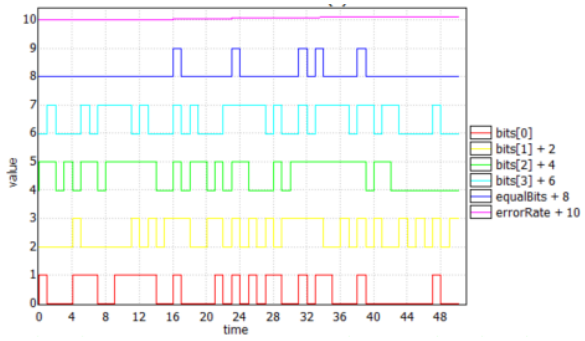
The first test will check if the system works correctly. For that, I will use both templates that generate inputs. The two next plots are generated in the verifier tool on UPPAAL with the next command:

`simulate[<=`*bound*`]{bits[0],bits[1]+2,bits[2]+4,bits[3]+6,equalBits+8, errorRate+10}`

*bound* is set in 16 for the first simulation and in 50 for the second.




(a) Simulation using `set_inputs`




(b) Simulation using `set_inputs_random`

Starting from below, the variables shown are the 4 inputs, `equalBits`, that says if there has been a failure in the iteration, and the error rate.
In figure 5.4a we can see that all possible inputs have been generated. The first one is the only one that fails, so the error rate is 1 at that moment. Then, it decreases in every iteration, reaching 0.0625 at the end of the simulation, the theoretical error rate value.
In figure 5.4b the inputs have a random distribution, so the error rate change depending on them.
With this information, it can be concluded that **the system is correct**.

### Error rate average

The template that generates random inputs will be used. With it, I will generate $N$ random inputs, and check the error rate in each simulation. Finally, I will test, with the values obtained, if we can consider that the real mean of the system is 0.0625. It will be used the same command of the first test, changing the *bound* depending on $N$. These are the error rates calculated:

| N | Error Rate |
|---|---|
| 10 | 0.1% |
| 50 | 0.02% |
| 100 | 0.0606% |
| 200 | 0.08% |
| 300 | 0.074% |
| 400 | 0.0625% |
| 500 | 0.076% |
| 750 | 0.053% |
| 1000 | 0.077% |
| 2000 | 0.069% |

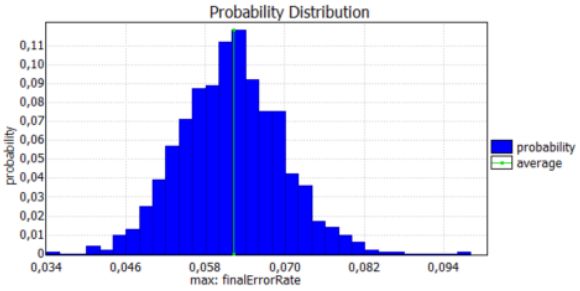
Now, the same hypothesis test done in the first implementation will be performed with this data. The statistical parameters are $\overline{X} = 0.06721$ and $s = 0.02097027$.With them, the statistical $t_{obs}$ is calculated: $t_{obs} = \frac{0.06721-0.0625}{0.0297027}\sqrt{10} = 0.710259$. This statistical follows a distribution $t_9$. Now, I calculate the p-value of the test, that is $2 * Pr(t_9 > 0.710259) = 0.4955$, which means that the null hypothesis $H_0 : \mu = 0.0625$ can not be rejected, so the same result of the first test has been obtained.

### Behaviour of the error rate

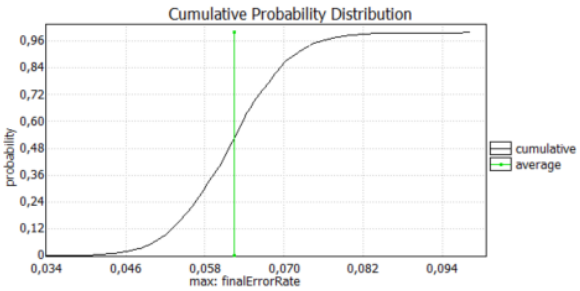
In this test I will obtain the statistical behaviour of the error rate, calculating the expected value of the maximum of the variable `finalErrorRate`. It will be done with the next command:

```
E[<=1000;1000](max:finalErrorRate)
```

Some of the plots obtained are:




(c) Probability Distribution Plot




(d) Cumulative Probability Distribution Plot

As numerical values, the mean of the 1000 simulations is a error rate of 0.0623, with a confidence interval ($\alpha = 0.05$) of $[0.061827, 0.062773]$, which include the theoretical error rate.
Observing the plots, we can conclude that the error rate follows a normal distribution with mean 0.0623.

### Comparison of the resources used by both multipliers

In this last test, I will check the time that the verifier tool last to complete $N$ iterations of each multiplier separately. I will disable `outputs_acc` or `outputs_aprox`, depending on the multiplier which will be tested. The command used on the verification of the model will be used, varying the *bound*. These are the times obtained, in seconds:

| N | Accurate multiplier | Approximate multiplier |
|---|---|---|
| 100 | 0.016 | 0.015 |
| 1000 | 0.046 | 0.031 |
| 2000 | 0.094 | 0.078 |
| 3000 | 0.125 | 0.093 |
| 4000 | 0.156 | 0.125 |
| 5000 | 0.172 | 0.156 |
| 6000 | 0.188 | 0.171 |
| 7000 | 0.219 | 0.203 |
| 8000 | 0.266 | 0.219 |
| 9000 | 0.297 | 0.266 |
| 10000 | 0.359 | 0.281 |
| 11000 | 0.391 | 0.296 |
| 12000 | 0.422 | 0.313 |
| 15000 | 0.453 | 0.375 |

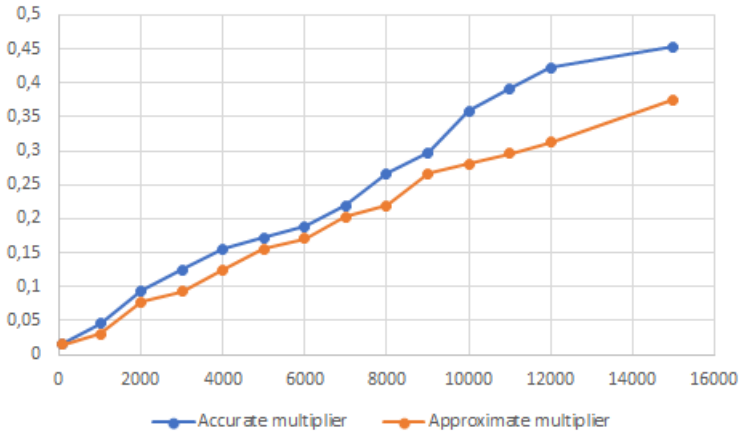



Figure 5.4: Plot of times of both multipliers

It seems that there are no such big differences between both times. To check it, I will do a regression test using R. The code used in that software for the analysis is the same as the used in the regression analysis done before, but changing the values of `y1` and `y2` for the obtained in this experiment. These are the outputs:

```
Call:
lm(formula = y1 ~ N)

Residuals:
      Min        1Q    Median        3Q       Max
-0.036902 -0.007620 -0.005168  0.010449  0.026150

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.096e-02  9.799e-03   2.139   0.0537 .
N           3.126e-05  1.239e-06  25.224 9.15e-12 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.01983 on 12 degrees of freedom
Multiple R-squared:  0.9815,    Adjusted R-squared:  0.9799
F-statistic: 636.2 on 1 and 12 DF,  p-value: 9.151e-12
```

Figure 5.5: Regression analysis results for the accurate multiplier

```
Call:
lm(formula = y2 ~ N)

Residuals:
       Min         1Q     Median         3Q        Max
-0.0195290 -0.0062263  0.0002981  0.0068019  0.0203883

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.224e-02  5.564e-03   3.996  0.00177 **
N           2.482e-05  7.038e-07  35.267 1.72e-13 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.01126 on 12 degrees of freedom
Multiple R-squared:  0.9904,    Adjusted R-squared:  0.9896
F-statistic:  1244 on 1 and 12 DF,  p-value: 1.725e-13
```

Figure 5.6: Regression analysis results for the approximate multiplier

The first analysis has a high $R^2$ and a small p-value, which means that the results are significant. The regression line is $y1 = 0.002096 + 0.00003126N$. In this case, it is not convenient to delete the intercept on the model, because it's p-value, although it is greater than 0.05, is very close to this value.

In the second analysis, we also obtain that the model is significant. The regression line is defined by $y2 = 0.00224 + 0.00002482N$. This is the definitive regression line because the p-values of the parameters say that they are significant in the model.

Comparing both lines, the time of the approximate multiplier grows slower than the time of the accurate multiplier, but there is not such a big difference in it.

## 5.3 Comparison of the two implementations

In both implementation have been tested similar characteristics, obtaining similar results in all of them except on the last one, the test that checks the time that the approximate and the accurate multiplier last on doing $N$ iterations separately. In the test done on the gate network model, there was a notable difference between both multipliers, but that behaviour doesn't appear in the truth table model. In my opinion, this difference is caused because of the number of states that are in the first model. The accurate multiplier has more gates than the approximate one, so more transition needs to be done and more time is used. The second model has the same number of iterations in both multipliers, but there is 1 calculation less on the approximate one because it just needs to find 3 outputs and not 4 as the accurate multiplier does. This could cause the insignificant difference between them.

In conclusion, observing all the results, it has been proved that **the truth table implementation is more efficient than the gate network one**.

## 5.4 Tests on DRAM model

For all the tests, the dimension of the DRAM will be 1 row and 8 columns, but it can be increased as much as the user wants.

### Verification of the system

As it has been done before, I will check that the model works correctly. I will write in some registers with different times of refresh and see what are the output. For the test, I will use the template shown in 4.16. These are two plots with different time of refresh:

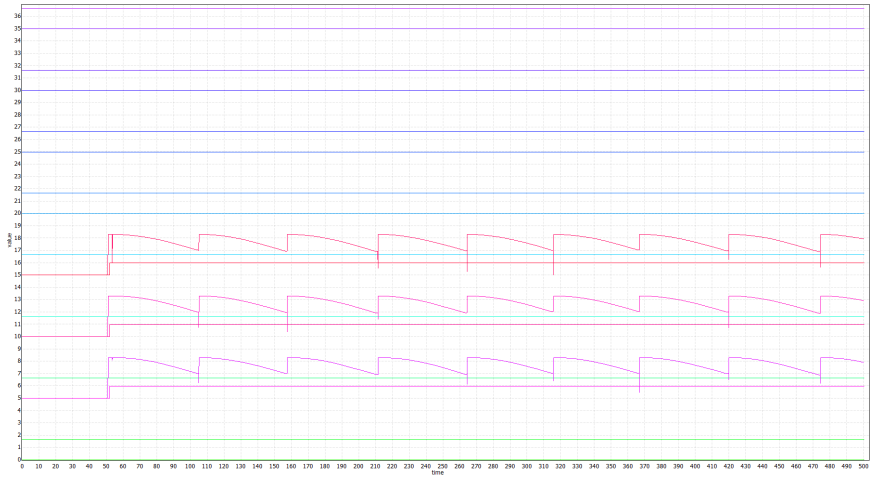


Figure 5.7: Simulation with `tRefresh` = 50

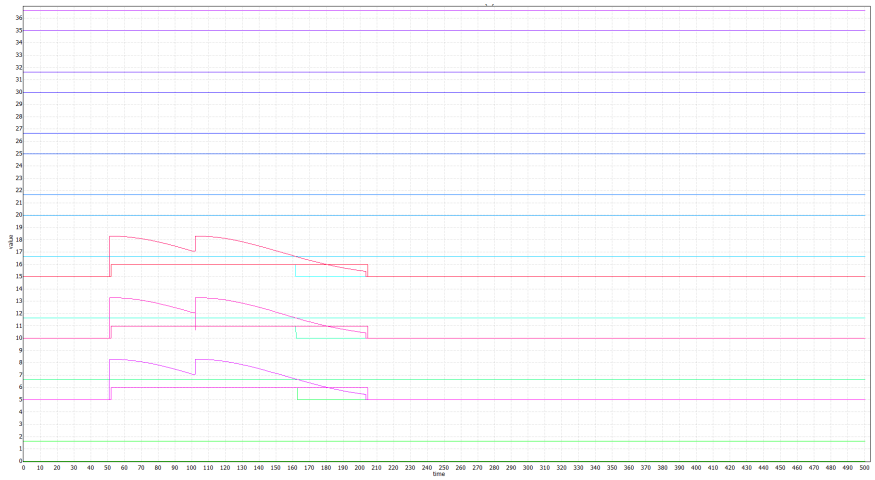


Figure 5.8: Simulation with `tRefresh` = 100

Each plot is divided into 8 parts as if they were the 8 columns of the DRAM. Each part has 4 variables on it: `cVolt` representing the voltage, `V_TRESH` representing the threshold, `eBit` representing the expected logical value and `cBit` representing the real logical value.
In both figures, there is a writing on the cells 1,2 and 3 because the data introduced was 2,4 and 8. The rest of the cells stays doing nothing.
In figure 5.7, the time of refresh is enough so the voltage never gets the threshold, so the real value is equal to the expected value all the time.
In figure 5.8, the voltage exceeds the threshold before the second refresh, so the real value is not equal to the expected value anymore. When a refresh is done, the real value is 0, so it doesn't recharge again.
Therefore, we can conclude that **the model works correctly**.

### Error rate depending on the time of refresh

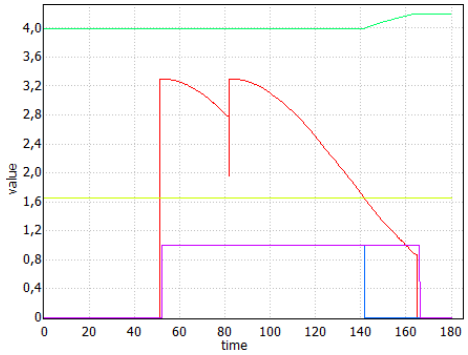
With this test, it is wanted to be known what is the error rate of the system varying the time of refresh of the system. Only 1 cell will be used because a writing in several cells at

the same time returns the same error rate on all of them. The test template used is similar to figure 4.16. There is one change on it: The argument of the function *write8* now is 1 instead of (2+4+8). This template simulates a writing on cell 0 on the time unit 50. The command used in UPPAAL verifier is:

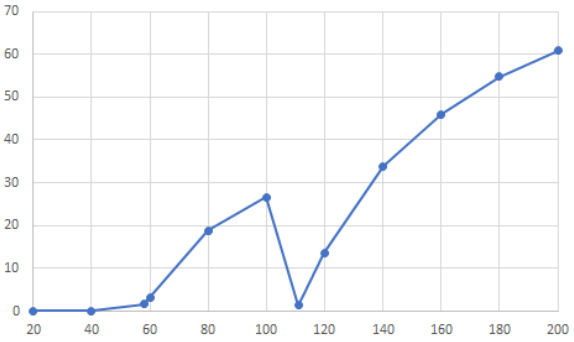
```
simulate[<=bound]{cVolt[0][0],eBit[0][0],
cBit[0][0],V_TRESH,errorRate[0][0]+4}
```

*bound* should be as big as necessary to see the correct error rate.
These are the results:

| Time of refresh | Number of refreshes before a failure | Error rate |
|---|---|---|
| 20 | - | 0% |
| 40 | - | 0% |
| 58 | 1 | 1.59% |
| 60 | 1 | 3.2% |
| 80 | 1 | 18.8% |
| 100 | 1 | 26.57% |
| 111 | 0 | 1.41% |
| 120 | 0 | 13.63% |
| 140 | 0 | 33.82% |
| 160 | 0 | 46.05% |
| 180 | 0 | 54.72% |
| 200 | 0 | 60.83% |




(a) Simulation with `tRefresh` = 80




(b) Error rate depending on the time of refresh

In plot 5.9a the writing is correct till the time 140 approximately, where the voltage starts to be lower than the threshold and the expected value is set on 0. There is a refresh on time 80, that can be seen because the voltage value is again the maximum possible.
In plot 5.9b we can see that the error rate always increases with the time of refresh, except on one point, on 111. This is because on that point there is a refresh just after the threshold has been reached, so the system updates its real value when very little time has passed since the expected value was updated.
In this case, if the system allows committing some failures, then **the time of refresh that should be selected is 111** because it is the time which gets a lower error rate and without doing any refreshes. If there would be an error rate even lower but doing one refresh , then

the system manager must decide what is the best for him, get less error rate but doing a refresh, which has a cost, or gets a little more error rate and doesn't do any refreshes.
This results will change depending on the experiment done. This is given because the writing has been done on the time unit 50, but if this operation is done at another moment, the optimal time of refresh will change. This is caused because the time of refresh is done all times at the same moment, doesn't matter when an operation happens.

# Chapter 6

# Conclusion

The aim of this thesis was to prove that approximate computing systems are useful if some errors can be accepted in the results obtained. For that, two types of models have been designed, implemented and tested: Two models of accurate and approximate logical multiplier and a model of a dynamic random access memory (DRAM). These implementations have been done using UPPAAL SMC. It has been a really useful tool, that allows building the models in an intuitive way.

Various versions of the models were created during the process, and some problems have also occurred, like one with the simulation tool of UPPAAL that always generate the same random numbers, so any tests could be done with it. Also, some problems with the set of time. This was my first time using this program and I had a hard time getting used to it.

In my opinion, the final results obtained on the different tests performed confirm that the approximate computing systems have a lot of utilities in different fields, such as in image and sound processing, machine learning or artificial intelligence, where some bits can be lost without having serious problems.

Some other tests could be done on these systems but are not implemented in this thesis could be amplify the number of inputs and outputs of the multipliers and see if the error rate is still as low as it is with 4 inputs, or check the behaviour of the error rates with more writings on different moments on the DRAM model.

# Bibliography


[1] David, A.; Gjøl, P.; Larsen, K. G.; et al.: *UPPAAL STRATEGO*. 2015. [Online; visited 09.05.2019].
Retrieved from:
https://link.springer.com/chapter/10.1007%2F978-3-662-46681-0_16

[2] David, A.; Larsen, K. G.; Legay, A.; et al.: *UPPAAL SMC tutorial*. 2015. [Online; visited 09.05.2019].
Retrieved from: https://doi.org/10.1007/s10009-014-0361-y

[3] Emerging Computing Technology Laboratory at SJTU: *Approximate Computing*. [Online; visited 09.05.2019].
Retrieved from: http://umji.sjtu.edu.cn/~wkqian/research.html

[4] Kugler, L.: *Is 'good enough' computing good enough?* 2015. [Online; visited 09.05.2019].
Retrieved from: https://cacm.acm.org/magazines/2015/5/186012-is-good-enough-computing-good-enough/fulltext

[5] Legay, A.; Delahaye, B.; Bensalem, S.: *Statistical Model Checking: An Overview*. 2010. [Online; visited 09.05.2019].
Retrieved from: https://arxiv.org/pdf/1005.1327.pdf

[6] Plasma Lab: *Statistical Model Checking*. [Online; visited 09.05.2019].
Retrieved from:
https://project.inria.fr/plasma-lab/statistical-model-checking

[7] PowerData: *Big Data: ¿En qué consiste? Su importancia, desafíos y gobernabilidad*. [Online; visited 09.05.2019].
Retrieved from: https://www.powerdata.es/big-data

[8] Sampson, A.: *Hardware and Software for Approximate Computing*. 2015. [Online; visited 09.05.2019].
Retrieved from:
https://digital.lib.washington.edu/researchworks/bitstream/handle/1773/33693/Sampson_washington_0250E_14938.pdf?sequence=1

[9] UPPAAL: [Online; visited 09.05.2019].
Retrieved from: www.uppaal.org

[10] UPPAAL: *Locations*. [Online; visited 09.05.2019].
Retrieved from:

http://www.it.uu.se/research/group/darts/uppaal/help.php?file=System_Descriptions/Locations.shtml

[11] UPPAAL: *UPPAAL TIGA*. [Online; visited 09.05.2019]. Retrieved from: http://people.cs.aau.dk/~adavid/tiga/index.html

[12] UPPAAL: *UPPAAL 4.0: Small Tutorial*. 2009. [Online; visited 09.05.2019]. Retrieved from: http://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf

[13] Wikipedia: *Approximate computing*. [Online; visited 09.05.2019]. Retrieved from: https://en.wikipedia.org/wiki/Approximate_computing

[14] Wikipedia: *DRAM*. [Online; visited 09.05.2019]. Retrieved from: https://es.wikipedia.org/wiki/DRAM

[15] Wikipedia: *Dynamic Random Access Memory*. [Online; visited 09.05.2019]. Retrieved from: https://en.wikipedia.org/wiki/Dynamic_random-access_memory

[16] Yoon, A.: *Understanding Memory*. 2018. [Online; visited 09.05.2019]. Retrieved from: https://semiengineering.com/whats-really-happening-inside-memory/