Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
(Mención en Computación)

Design and Implementation of Approximate Computing Systems

Autor: **D. Sergio Pérez Hernández**

Tutor: **D. Quiliano Isaac Moro Sancho**

Resumen

Este trabajo se centra en el diseño y creación de sistemas computacionales aproximados, los cuales aceptan una determinada tasa de error para mejorar otras características, como la velocidad, o reducir costes. Para este proceso se utilizará la herramienta UPPAAL SMC.

Abstract

This thesis is focused on the design and implementation of approximate computing systems. This kind of systems allows some errors but improving other aspect, such as speed, or reducing resources. For this proccess, UPPAAL SMC will be used.

Acknowledgements

I would like to than Mr. Josef Strnadel, teacher of the Information Technology Faculty at Brno University of Technology,for his effort and dedication in helping me throughout the proccess of creating my thesis.

Contents

Chapter 1

Introduction and objectives

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.([5])
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.([12])

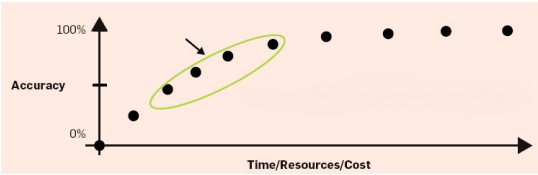


Figure 1.1: Approximate computing accuracy plot ([4])

Objectives

In this Bachelor's thesis, the objectives to complete are:

- Design some models of approximate computing systems, which will be a logic multiplier and a DRAM system.
- Implement the proposed solutions in UPPAAL 4.1, which is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata([8]).

- Check the correctness of the systems, observing if the requirements of the problems have been solved.
- With several simulations of these implementations, obtain some data that I will use in my Statistics Bachelor's thesis (6) to check if an approximate computer system can be more efficient than an accurate one, using also UPPAAL 4.1.

Report Content

This memory will be divided in four different parts:

1. Background: I will explain what approximate computing is, its utilities and where it is used.
2. Implementation areas and Means: Describes the models implemented and the basics about UPPAAL 4.0 and 4.1
3. Proposed solution: Shows how the implementation has been done and its characteristics.
4. Conclusions: I will give my impressions, difficulties and possible improvements of this job.

Time Planning

The time that I will expend into this thesis will be:

- Two weeks to search for information about the background and UPPAAL.
- One month to learn how to use UPPAAL 4.1, because this will be my first time using this tool.
- Two months to design and implement the proposed systems.
- Two weeks to look for errors in the implementation, fix them and obtain some data.

With this estimation, this thesis is expected to be done in around 4 months.

Chapter 2

Background: Approximate Computing

Approximate computing is a research agenda that seeks to better match the accuracy in system abstractions with the needs of approximate programs ([7]). 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 involved 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.

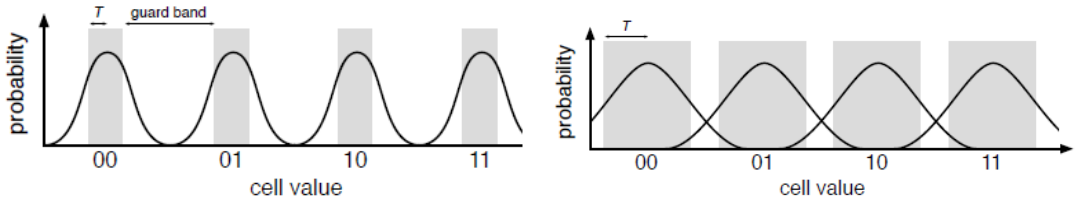
Interfaces for approximate storage: Modern non-volatile memory technologies exhibit properties that make them candidates for storing data approximately. By exploiting the synergy between these properties an application-level error tolerance, we can alleviate some of these technologies' limitations: limited 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 ([7])

(b) Approximate MLC ([7])

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.

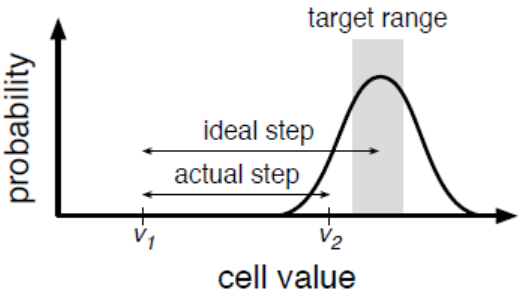


Figure 2.1: Single step in an iterative program-and-verify write ([7])

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 design and implement two approximate logic multipliers, while in the second I will build a DRAM memory using UPPAAL 4.1. Then, I will evaluate the behaviour of this implementations in my Statistics thesis (6).

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 error tolerant. 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

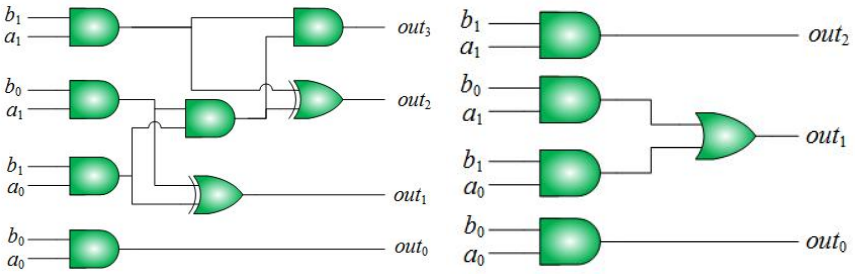
Table 3.1: Truth table of the accurate logical multiplier

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.(13; 14)
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.

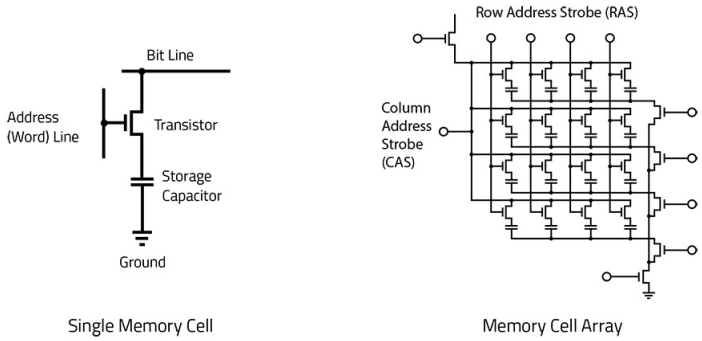


Figure 3.1: DRAM scheme (15)

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 (11).
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, which means that, in each transition, a clock will increment its value in 1 (with some exceptions explained below). 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.

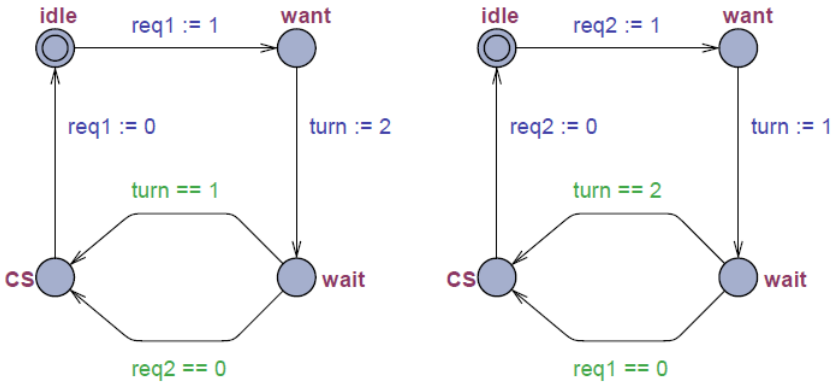


Figure 3.2: Example of a UPPAAL model (11)

Locations

There are different kinds of locations of UPPAAL (9):

- **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.

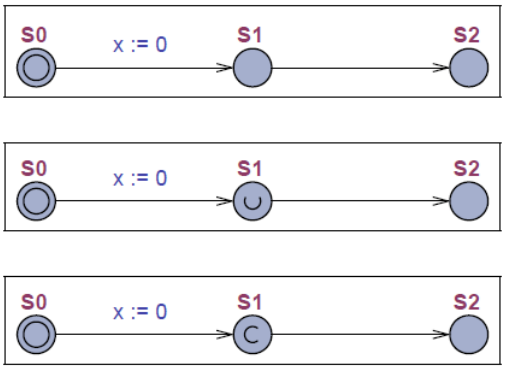


Figure 3.3: Example of a normal, urgent and committed states

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.

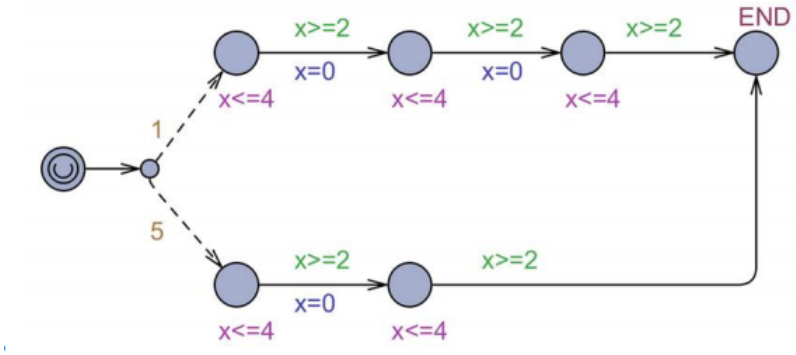


Figure 3.4: Example of a stochastic timed automata ([2])

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.

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 ([10]).
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.([11])

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.

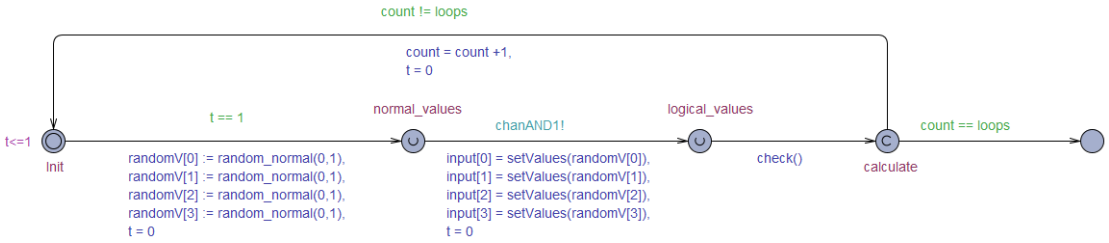


Figure 4.1: Main template

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.

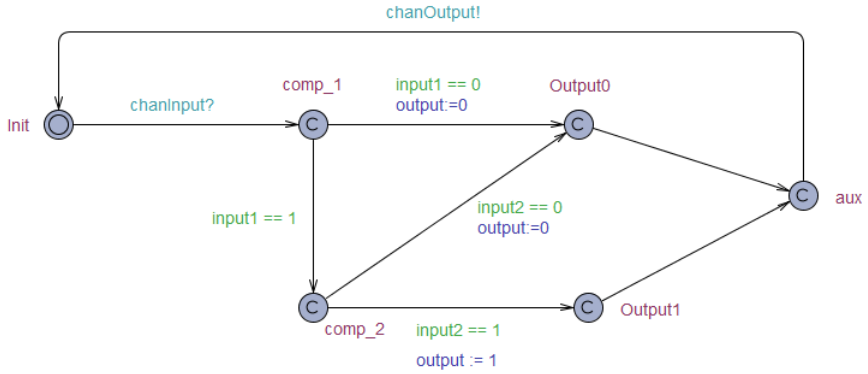


Figure 4.2: AND template

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.

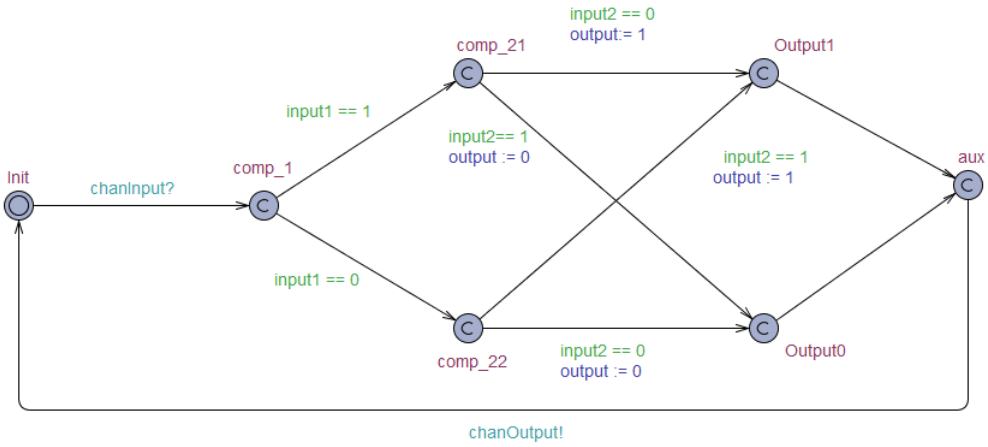


Figure 4.3: XOR template

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.

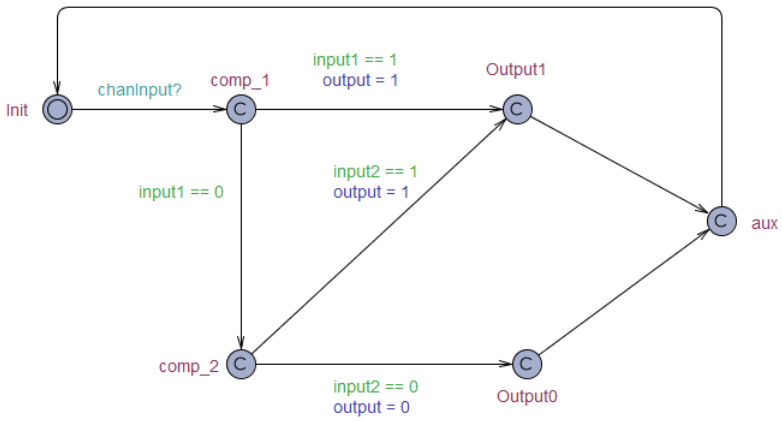


Figure 4.4: OR template

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()*

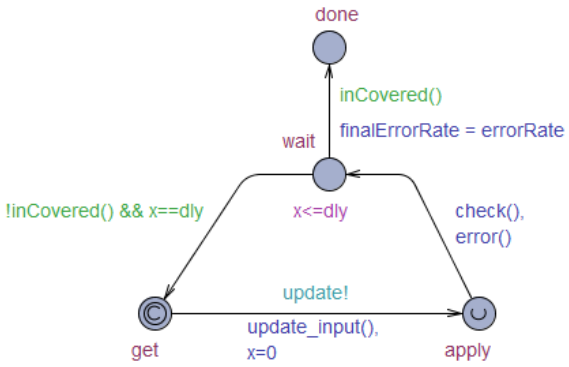


Figure 4.8: `set_inputs` and `set_inputs_random` template

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.

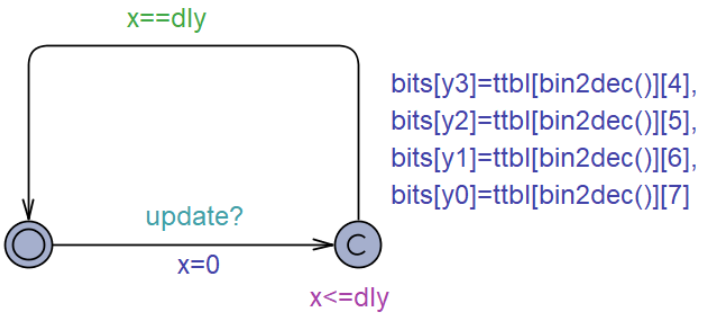


Figure 4.9: `outputs_acc` template

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()*.

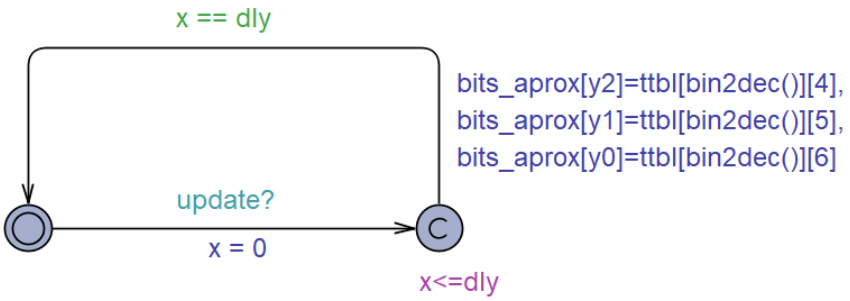


Figure 4.10: `outputs_aprox` template

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:

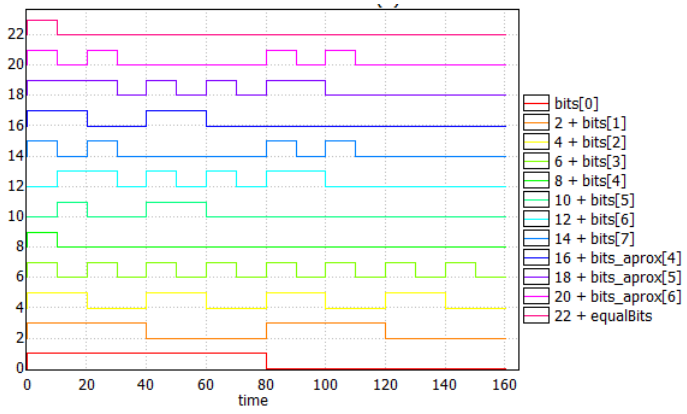


Figure 4.11: Simulation with `get_inputs`

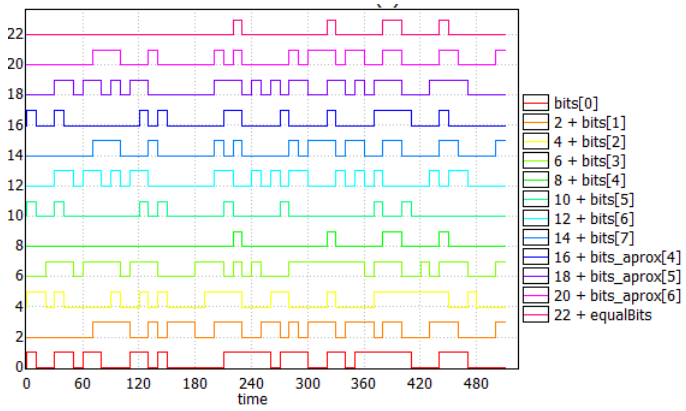


Figure 4.12: Simulation with `get_inputs_aprox`

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.

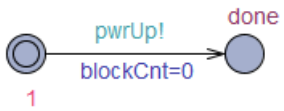


Figure 4.13: `powerUp` template

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.

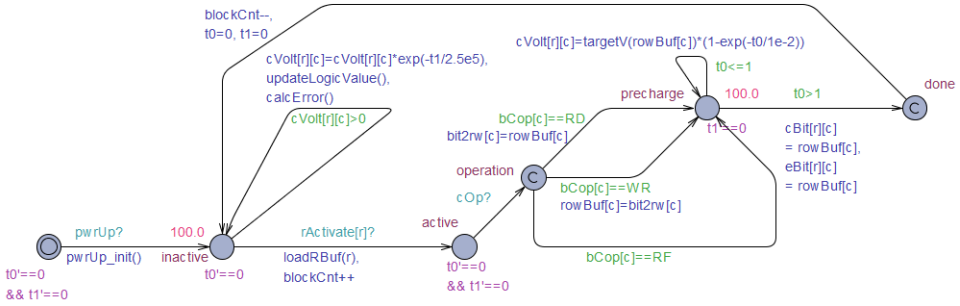


Figure 4.14: `mCell` template

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.

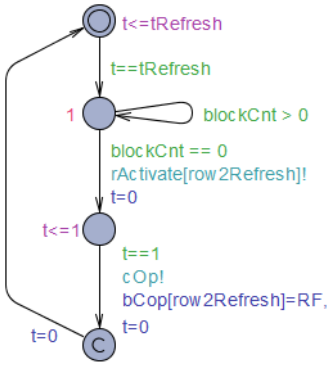


Figure 4.15: `mRefresh` template

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.

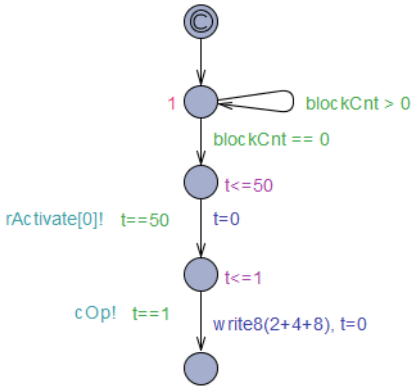


Figure 4.16: `test` template

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:

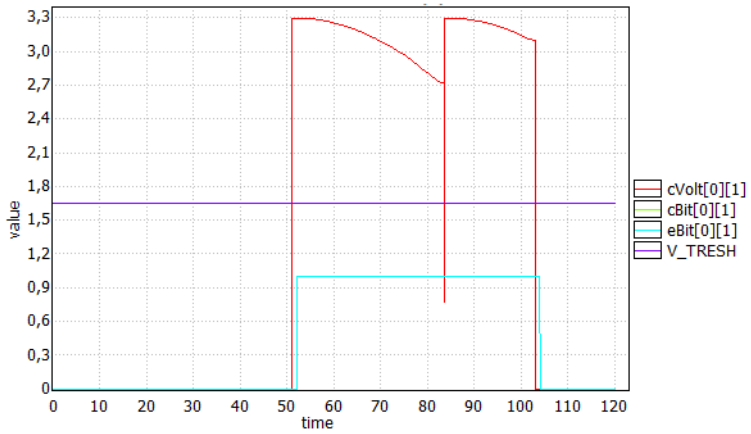


Figure 4.17: DRAM simulation example

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

Conclusions

In this thesis, I have designed and developed three approximate computing system models: Two logical multipliers using two different approaches and a Dynamic Random Access Memory (DRAM). These models have been created 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. One of these problems that I had to manage with was the set of time, that made the creation of the models more difficult than I thought at the beginning.This was my first time using this program and I had a hard time getting used to it, so I took more time than expected in learning the basics and some advances methods. First I started reading the specifications of the tool and doing some easy examples. Then I created more difficult sequences till I could finally do the implementations that I have presented in this thesis.

Some other implementations could be done, such as change the number of inputs of the logical multipliers or improve the functionality of the DRAM system. These tasks can be done in the future.

Executing these programs, I will get some data that will be used in my Statistics Bachelor's thesis ([6]) to see its behaviour and check if approximate computer systems have any advantages over the accurate ones.

Bibliography

[1] David, A.; Gjøl, P.; Larsen, K. G.; et al.: *UPPAAL STRATEGO*. 2015. [Online; visited 11.02.2020].
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 11.02.2020].
Retrieved from: https://doi.org/10.1007/s10009-014-0361-y

[3] Emerging Computing Technology Laboratory at SJTU: *Approximate Computing*. [Online; visited 11.02.2020].
Retrieved from: http://umji.sjtu.edu.cn/~wkqian/research.html

[4] Kugler, L.: *Is 'good enough' computing good enough?* 2015. [Online; visited 11.02.2020].
Retrieved from: https://cacm.acm.org/magazines/2015/5/186012-is-good-enough-computing-good-enough/fulltext

[5] PowerData: *Big Data: ¿En qué consiste? Su importancia, desafíos y gobernabilidad*. [Online; visited 11.02.2020].
Retrieved from: https://www.powerdata.es/big-data

[6] Pérez, S.: *Statistical Model Checking in Approximate Computing Systems*. 2020.

[7] Sampson, A.: *Hardware and Software for Approximate Computing*. 2015. [Online; visited 11.02.2020].
Retrieved from: https://digital.lib.washington.edu/researchworks/bitstream/handle/1773/33693/Sampson_washington_0250E_14938.pdf?sequence=1

[8] UPPAAL: [Online; visited 11.02.2020].
Retrieved from: www.uppaal.org

[9] UPPAAL: *Locations*. [Online; visited 11.02.2020].
Retrieved from: http://www.it.uu.se/research/group/darts/uppaal/help.php?file=System_Descriptions/Locations.shtml

[10] UPPAAL: *UPPAAL TIGA*. [Online; visited 11.02.2020].
Retrieved from: http://people.cs.aau.dk/~adavid/tiga/index.html

[11] UPPAAL: *UPPAAL 4.0: Small Tutorial*. 2009. [Online; visited 11.02.2020].
Retrieved from: http://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf

[12] Wikipedia: *Approximate computing*. [Online; visited 11.02.2020].
Retrieved from: https://en.wikipedia.org/wiki/Approximate_computing

[13] Wikipedia: *DRAM*. [Online; visited 11.02.2020].
Retrieved from: https://es.wikipedia.org/wiki/DRAM

[14] Wikipedia: *Dynamic Random Access Memory*. [Online; visited 11.02.2020].
Retrieved from: https://en.wikipedia.org/wiki/Dynamic_random-access_memory

[15] Yoon, A.: *Understanding Memory*. 2018. [Online; visited 11.02.2020].
Retrieved from: https://semiengineering.com/whats-really-happening-inside-memory/