TUHH
Hamburg University of Technology

Project Thesis

Salvador Fernandez Covarrubias

Structure and parametrization learning for probabilistic timed automata

June 1, 2018

supervised by:
Prof. Dr. Sibylle Schupp
M.Sc. Sascha Lehmann

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

STS
Software
Technology
Systems

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 1. Juni 2018

Salvador Fernandez Covarrubias

Contents

List of Figures

List of Algorithms

Abstract - It is common in many experimental situations, to derive models from observed systems, which can be used to simplify and predict the behavior of data. Such models may need to be solely derived from the output data of observed systems, due to the lack of unknown specified behavior. In this project thesis, we design and implement a framework that involves the derivation of such models. To accomplish this, we identify equations that best describe the incoming data of a system via function fitting, transform them into model locations with an assigned probabilistic weight, based on the similarities and occurrences of the identified equations in multiple traces, and finally merge the locations to a probabilistic model. In two experiments, we demonstrate the capability of the framework to recreate models from overlapping data of traces, and from the data of a cyclic heart beat simulation, which is highly sensitive to the parameters of the framework.

1 Introduction

Nowadays, it is possible to model and verify the properties of real-time systems with the use of software. Model checking has been broadly used in engineering to ensure the correctness of the functionality of a constructed model. The aim of this project is to recreate a model through the analysis of its output data. It means that the data obtained from a model simulation, is going to be an input for another system that will analyze it. The scenario can be explained as if a system attempts to teach its functionalities to another system. However, a problem arises when we represent the functionality of a system by only analyzing data.

Many systems may have non-deterministic behaviors that provide unpredictable outputs, and complicate the learning process for other systems. Let us imagine a machine with the functionality of tossing coins and registering whether the coin landed with a *heads* or *tails* facing up. The previous functionality can be represented by constructing a model whose sole action is to output data that is either *heads* or *tails*. Now let us attempt to reconstruct the functionality of the machine by only analyzing a specific output from it. Assuming that the machine threw a coin 20 times and that the result was always *heads*, we are able to represent a model that only registers *heads* as an outcome because we analyzed data that contains the same information (*heads*). But now let us assume that the machine threw 20 coins and that half of the results were *heads*. It is only possible with the analysis of the previous information, to model a perfectly random coin-thrower machine. It is possible to recreate the functionality of a non-deterministic model, which is why this research paper focuses on demonstrating how the output of a system can be represented as a probabilistic model that resembles its functionality. This thesis begins by introducing some basic concepts of the theory of timed automata and UPPAAL in Chapter 3. Next, we explain how data is analyzed and transformed to a set of equations in Chapter 4 and Chapter 5. Afterwards, we discuss the approach of representing equations as probabilistic models in UPPAL in Chapter 6 and Chapter 7. In Chapter 8 we explain how the functionality of a recreated model resembles the data from which it was created. Then we take a look at some experiments of data that was successfully represented as probabilistic models in Chapter 9. The future work is discussed in Chapter 10, and the conclusion that sums up the essence of model recreating in Chapter 11.

2 Related Work

There are two main environments in which a system can learn from the observation of data. The first one is an *on-line* approach [5], in which a system is able to identify a model by observing data from another system in real time. The second one is an *off-line* approach [5], that allows a system to identify a model by consulting previously stored data (e.g. data from a database). Despite the fact that both approaches interact very differently with data, they are both capable of identifying the functionalities of a model through the analysis of information. We discuss in this project how a model can be recreated with the use of an *offline* approach.

The analysis of data is performed by using a *function fitting* approach, to recreate a system as a probabilistic models in UPPAAL. The *function fitting* that we use is based on the principle of *data fitting* [3], which has been used in many fields to fit a model or mathematical function to a set of data. The main objective is to capture the trend of the data by assigning a function across a specified range of values. Every function calculated with the previous approach is treated in this project as a $fittedFunction$, which is an estimation of the overall behavior of a set of data.

Model optimization is a very wide subject applied in many fields that consists of the elimination of redundant information, with the aim of reducing the complexity of a model. The TTT algorithm [4] reduces the complexity of a model by analyzing counter examples of the same model that expose the use of redundant information. We apply a similar concept by analyzing the trend of every $fittedFunction$ in real time, with the purpose of creating a minimal probabilistic model structure.

The motivation of this paper comes from the fact that a model can be recreated by using many approaches. The aim is to demonstrate that there exists a way of recreating a model by the analysis of its traces. Our implementation is limited to an *off-line* approach, but the same technique can be applied to other ones.

3 Background

We will review in this section the main concepts that help to understand the process of recreating a model. We start with an introduction of theory of automata and timed automata. Then we explain the most relevant features of UPPAAL that allows to represent timed automata as models.

3.1 Theory Of Automaton

Generally speaking, an automaton is a mechanism or machine that automatically executes a sequence of operations.

Definition 3.1.1. (Automaton) An automaton A is a 5-tuple $< Q, \mathcal{E}, E, q_0, F >$ [2] where,

- Q is a finite set of states.
- $\mathcal{E}$ is a finite set of symbols that are part of the alphabet of A.
- δ is the transition function: Q x $\mathcal{E} \to Q$.
- q_0 is the initial state; $q_0 \in Q$.
- F is a set of states of Q; $F \subseteq Q$.

3.2 Theory Of Timed automaton

A timed automaton is an extension of an automaton with clock variables, that represent an abstract model of a real-time system.

Action It is a finite set of symbols $\mathcal{E}$ from an alphabet that represent an action.

Clock Constraints It is a boolean expression $B(C)$ that evaluates a finite set of clocks C and restricts actions.

Remark. Clock constraints are mostly used as guards and invariants for locations.

Definition 3.2.1. (Timed Automaton) A timed automaton A is a tuple $< N, l_0, E, I >$ [1] where,

- N is a finite set of locations or states.
- l_0 is the initial location. $l_0 \in l$.
- $E \subseteq l$ x $B(C)$ x $\mathcal{E}$ x 2^C x l is the set of edges or transitions between locations.

– $I : l \rightarrow B(C)$ is the set of invariants assigned to a location.

A transition from one location to another is defined as $l \xrightarrow{g,a,r} l'$, where r is as an action that resets or reassigns the value of a clock.

3.3 The UPPAAL Modelling Tool

UPPAAL is a tool that allows to model, simulate and analyze the behavior of systems. It is used in this project for modelling and interacting with real-timed systems. We will only discuss the main components of UPPAAL models, the process of performing simulations, and a practical example that shows the structure of the information retrieved from a simulation.

3.3.1 Templates

A template is a structure that represents an automaton as a model, and that allows to declare parameters or variables to be used. There are many variable types that can be declared, but this project only utilizes the following:

- *Integer* Variable that store the values of whole numbers (not fractional).
- *Double* Variable that store the values of fractional numbers.
- *Clock* Variable that represents the time units of a model (internally recognized as a *Double* in UPPAAL).

3.3.2 Locations

In UPPAAL the states of a system are modelled as locations. The main properties of a location are:

- *Name* Used as a unique identifier for each location
- *Invariant* Boolean expression that compares the value of a clock variable against an integer to determine the active time of a location.
- *Type*
  - *Initial* Feature of a location that indicates the starting state of the system.
  - *Urgent* Feature that does not let time pass by in a location
  - *Committed* Feature that does not let time pass by in a location, and that also restricts the transition of the location to only outgoing edges from the current committed location.

3.3.3 Edges

Transitions that allows a location to travel to other locations or to itself, where each transition can be combined with a set of actions or conditions like the following:

- Guard
  - Boolean expression or condition that has to be satisfied in order to allow a location to execute a transition or edge.
- Update
  - Expression(s) or action(s) that can modify the values of declared variables from a template.

3.3.4 Branch Points

A branch point is a structure that is able to assign a probability-weighted value to an edge. It determines the probability of success of a location to execute an edge among all the outgoing edges from a branch point.

3.3.5 The Query language

UPPAAL has a verifier that allows the user to interact with the properties of a model, and to also retrieve data or *traces* from it through the use of *Queries*. The term *trace* represents a set of values of a variable (e.g. a *Double* variable) or component from a template (e.g. a location). In order to retrieve a trace from a model, it is necessary to perform the following query:

$simulate\ x\ [expression]\ \{variables\}$

where x is the number of rounds that the simulation will be executed, *expression* is a boolean condition that will determine the duration of the simulation and *variables* a set of variables of the template to be observed.

Remark. The duration of the simulation directly refers to a default global clock from UPPAAL that keeps track of the time from the simulation.

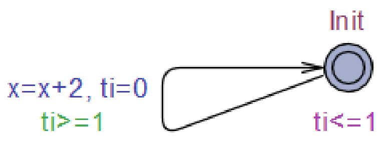
The execution of such query outputs x number of traces, that contain the values of each specified variable on *variables* from the beginning of the simulation until the specified duration time, given by *expression*.

An already existent tool implemented by Jonas Rinast was used in this project for loading UPPAAL models to execute queries, and to retrieve traces to further analyze them.

3.3.6 UPPAAL Model Example

Given an UPPAAL model or template declared as *Increment*:

Figure 3.1: Practical example

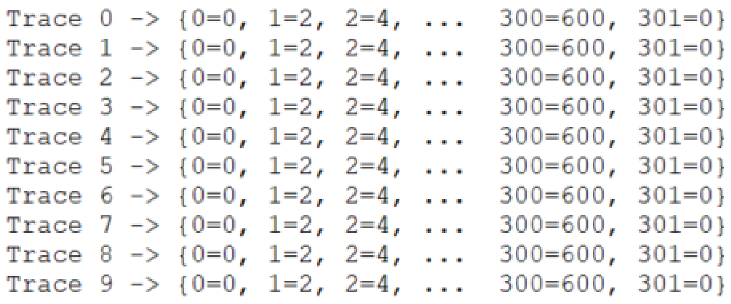


Init is an initial location (denoted by the circle inside of the location), that has an invariant $ti <= 1$, where ti is a declared clock variable. There is a self-edge on *Init*, that has a guard $ti > -1$, and an update action $x = x + 2, ti = 0$.

In summary, the model increments the value of x by 2 every time the value of ti changes from 0 to 1. We can create a simulation of the previous model by first loading it to the Model Generator tool and then executing the following query:

$simulate\ 1\ [<= 300]\ \{Increment.x, Increment.init\}$

Figure 3.2: Example of the extraction of a trace



```
Trace 0 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 1 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 2 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 3 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 4 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 5 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 6 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 7 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 8 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 9 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
```

The obtained traces are expressed as a set of assignments, which we will be calling *data points*. The left side of the assignment represents a time unit from the simulation, while the right side represents the value of the variable that was observed. The trace of variable x from template $Increment$ tells that x is initialized with a value of 0 at the first time unit of the simulation (0) and later incremented by 2 until the simulation ends (at time unit 300). The trace of variable *Init* (the initial location from the previous model), indicates with a $true$ value that the location was always active in the simulation. When a location is inactive in a simulation, the trace displays a $false$ value.

4 Model Recreation

We will discuss in this chapter an overview of how a model is recreated. The first step is to retrieve traces from a model in order to analyze them. The goal is to transform the data of the traces to a set of equations, that resemble a mathematical expression. Each equation describes the tendency of a trace and also represents a component of a probabilistic timed automata. After all of the equations are created, they are then gathered to conform an UPPAAL probabilistic model structure. Such model is referred in this paper as a *recreated model* because the source of its creation comes from the traces of another model. The aim is to recreate models only from traces of numerical variables, to compare the tendency of the data from *recreated models* against their source of data.

4.1 Methodology

The process of recreating a model based on its traces follows a specific set of procedures. First, the information from the traces has to be filtered by discarding any data from variables that are not numerical. Then a structure must be implemented in order to store the desired amount of data from the traces to be analyzed. Several types of analysis observe the data point values of a traces and fit the tendency of its data as a mathematical expression. The polynomial analysis fits the data to a polynomial function of degree n, the exponential analysis fits the data to an exponential function, and the harmonic oscillation analysis fits the data to a cosine function. Only the best fitted functions are transformed into equations, which will later conform a probabilistic model. We will only cover the details of the data preprocessing from traces in Section 4.2, the importance of a structure that controls the flow of the data in Section 4.4 and a general explanation of how the data is fitted to a function.

4.2 Trace Data Pre-Processing

The initial phase begins by retrieving a set of traces that we want to represent as a model. The traces are given by the model generator (explained in Section 3), which first needs to be filtered in order to proceed with the analyses. The filtering process of the information is described in Algorithm 1. The set of traces are stored in an array, where only the information of the data obtained from numerical variables (e.g. *Integer* or *Double* variables) is analyzed. We then proceed to initialize our data control structure called *buffer*, in order to analyze the desired trace and store the result on the list of *Analysis* objects. The functionality of the *buffer* and the $fitter$ method are discussed in the next sections.

Algorithm 1 Trace-analyzer

```
procedure ANALYZE(traceArray)
    for $i \leftarrow 1, traceArray.Size()$ do
        $trace \leftarrow traceArray[i]$
        for $j \leftarrow 1, trace.Size()$ do
            $traceData \leftarrow trace[j]$
            $varName \leftarrow traceData.getVariableName()$
            if $traceData.isNumeric()$ then
                buffer ← bufferInit(timeStep, bufferSize, traceData)
                analysis ← fitter(timeStep, buffer, varName, traceData)
                $traceAnalysisList.add$(analysis);
            end if
        end for
    end for
end procedure
```

4.3 The Analysis Class

It contains the results of all the analyses that were made to a trace. The *name* attribute represents the name of the variable that was analyzed, *initValue* the first value that the variable has at the beginning of a simulation, *traceId* is the id or number of the trace that was analyzed, and *results* a structure where all of the best fitted functions are stored as equations.

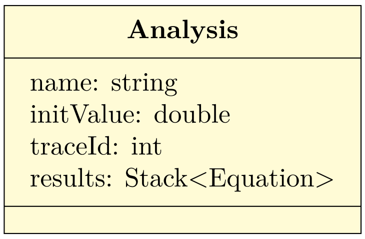
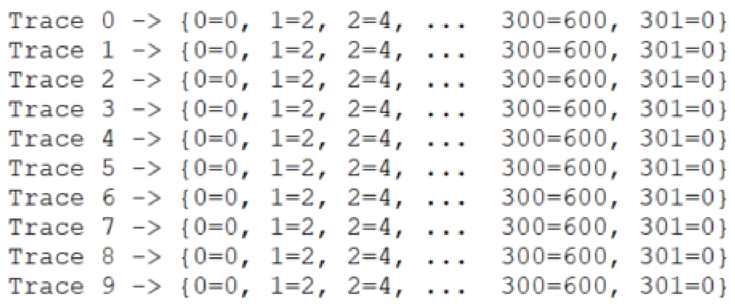


Figure 4.1: Structure of the Analysis class

4.4 Data Control Structure

The existence of a data control structure or *buffer* that retains data of a trace, arises from the necessity of controlling the quantity of data to be analyzed, and of how frequently it should be analyzed.



```
Trace 0 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 1 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 2 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 3 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 4 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 5 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 6 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 7 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 8 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
Trace 9 -> {0=0, 1=2, 2=4, ...  300=600, 301=0}
```

Reconsidering the previous example, it is noticeable that the trace has a size n of 301. A *buffer* is created by first choosing a size lower than n, which represents the quantity of data from the original trace that is going to be analyzed. A *time step* value must also be set, which tells how frequent the *buffer* is going to store new information to be analyzed. This last parameter allows the *buffer* to act as a sliding window, that will be shifting and retrieving data points from the original trace.

Remark. A *buffer* has the same structure as a trace because it is used to reference fractions of the data from it.

In the implementation of this project, the *buffer* is defined as a Java TreeMap class. Its main attributes are depicted in the figure below.

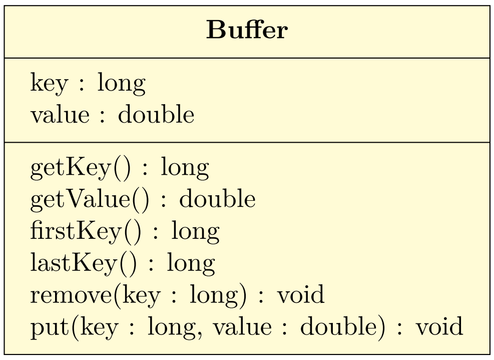
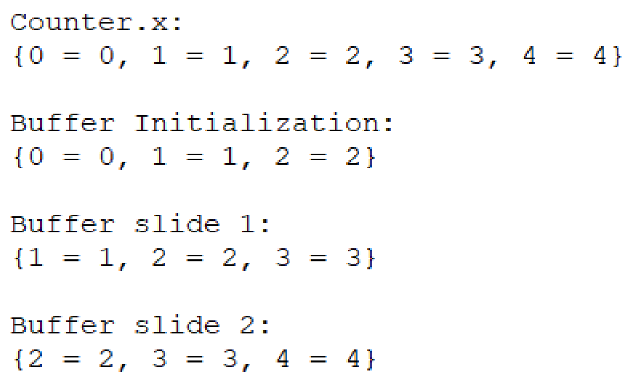


Figure 4.2: Structure of the Buffer class

The *key* attribute represents the time unit of a trace, *value* contains the value of the observed variable. There is a one-to-one relationship between *key* and *value*, which means that there cannot be repeated keys in a *buffer*.

We will now present an example that shows how the data of a trace is controlled by a *buffer* of size 3 and a time step of 1:

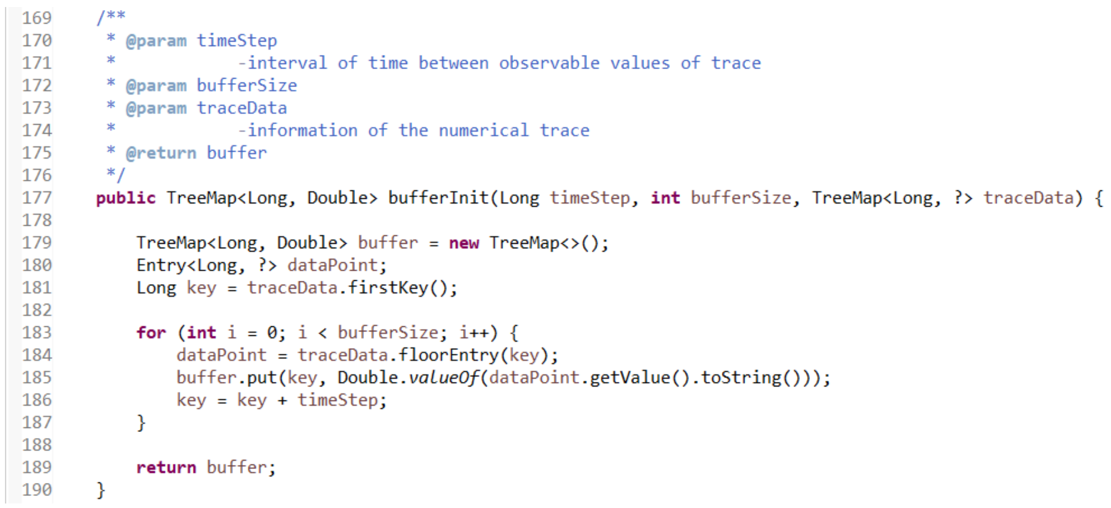
Figure 4.3: Example of a buffer of size 3 with a time step of 1



```
Counter.x:
{0 = 0, 1 = 1, 2 = 2, 3 = 3, 4 = 4}

Buffer Initialization:
{0 = 0, 1 = 1, 2 = 2}

Buffer slide 1:
{1 = 1, 2 = 2, 3 = 3}

Buffer slide 2:
{2 = 2, 3 = 3, 4 = 4}
```

We will use a section of the code from the implementation to explain the initialization of the *buffer*:

Figure 4.4: Method of the implemented tool for initializing a buffer



```
169  /**
170   * @param timeStep
171   *            -interval of time between observable values of trace
172   * @param bufferSize
173   * @param traceData
174   *            -information of the numerical trace
175   * @return buffer
176   */
177  public TreeMap<Long, Double> bufferInit(Long timeStep, int bufferSize, TreeMap<Long, ?> traceData) {
178
179      TreeMap<Long, Double> buffer = new TreeMap<>();
180      Entry<Long, ?> dataPoint;
181      Long key = traceData.firstKey();
182
183      for (int i = 0; i < bufferSize; i++) {
184          dataPoint = traceData.floorEntry(key);
185          buffer.put(key, Double.valueOf(dataPoint.getValue().toString()));
186          key = key + timeStep;
187      }
188
189      return buffer;
190  }
```

The first key of the *buffer* is obtained from the original trace (as seen in line 181). Each *value* that is inserted in the *buffer* is a reference from an entry *key* of the original trace. When the *buffer slides*, it means that a new entry is going to be inserted in it. Every entry to be inserted is retrieved from the original trace, by the use of the $floorEntry(key)$ method. It returns a key-value mapping associated with the greatest key less than or equal to the given key. The value of *key* is obtained by adding the value from the last key of the *buffer* and the value of *time step* (as seen in line 186). The value of *time step* is proportional to the amount of $shifts$ that the *buffer* does, and inversely proportional to the amount of data that is going to be analyzed from the trace.

4.5 Data Fitting

As previously mentioned, only the data stored in the *buffer* is fitted to one of the three types of functions. This section only focuses on demonstrating how the *buffer* is utilized to analyze all of the data points from a trace, and on how the best function is chosen among all the fitted ones. The methods *thresholdEvaluation* and *storeEquation* are discussed in Chapter 5.

Algorithm 2 Fitter

```
procedure FITTER(timeStep, buffer, varName, traceData)
    $lastEquationFits \leftarrow thresholdEvaluation($lastStackedEQ, buffer$)$
    if $lastEquationFits = true$ then
        $storeEquation(lastEquationFits)$
    else
        while $windowSlideLimit == false$ do

            for $i \leftarrow 0, buffer.Size()$ do
                $x \leftarrow buffer.getKey()$
                $y \leftarrow buffer.getValue()$
                $observedData.add(x, y)$
            end for

            $polynomialEquation \leftarrow$ fitPolynomial(observedData, varName)
            $exponentialEquation \leftarrow$ fitExponential(observedData, varName)
            $harmonicEquation \leftarrow$ fitHarmonic(observedData, varName)
            $bestFittedEquation \leftarrow$ min(fittedPolynomial, fittedExponential,
            fittedHarmonic)
            storeEquation(bestFittedEquation)

            $key \leftarrow$ buffer$.lastKey() + timeStep$
            buffer$.remove($buffer$.firstKey())$
            buffer$.put(key, traceData.floorEntry(key))$

            if buffer$.lastKey() > traceData.lastKey()$ then
                $windowSlideLimit \leftarrow true$
            end if

        end while
    end if
end procedure
```

The name of the variable to be analyzed is referenced by $varName$, x stores the time unit of a data point, and y stores the value of $varName$. Each data point extracted form the *buffer* is stored in an observation list called *observedData*. The list is passed as a parameter to three different types of fitting methods. After each methods finalizes its $fitting$ processes, an equation from each of them is stored in their respective variables: *polynomialEquation*, *exponentialEquation*, *harmonicEquation*. The equation with the lowest error is the one that best describes the behavior of the data, and also the one that is stored in the stack of fitted equations. After the best equation is stacked, the data of the *buffer* is *shifted*. The first key is removed and then the next data from the trace is added as the last key. It is always ensured that *buffer* never attempts to extract information from a trace that does not exist before repeating the whole process again. The algorithm ends when the last value of the original trace is reached by the *buffer*, which means that all of the data points of a trace have been successfully traversed. The details of how the data is fitted to a function are discussed in the next chapter.

5 Function Fitting Approach

As previously discussed, there are three types of analyses performed to the data points of a trace. Despite the fact that each analysis fits the data to different functions, all of them are treated as an equation that describes the tendency of the observed data from a trace. The previous procedure will be referred as $function\ fitting$. The next sections explain how the behavior of the data from a trace is identified and expressed as a function.

5.1 Polynomial Fitting

We first assume that the data points to be analyzed are actually coming from a polynomial function. As a polynomial function might be created with different coefficients and degrees. The purpose of the polynomial fitting algorithm is to attempt to fit the observed data points from a trace to a polynomial functions of degree n; where n is a natural number ($n = 0, 1, 2, 3, ...$). The structure of a fitted polynomial functions is as the following: $c_1 \pm c_2x \pm c_3x^2 \pm ....c_nx^{n-1}$; where c is a coefficient or constant of the function, and x a variable that represents the time unit of a trace. The degree of the function is determined by the highest exponential of its individual terms with non-zero coefficients.

5.1.1 Polynomial-Analyzer Algorithm

This algorithm utilizes the *PolynomialCurvFitter* class from the library *apache-commons-math3*. The objective is to a set of observed data to a polynomial function of degree n (given as a parameter) and returns the coefficients of the specified polynomial function. The procedure of the analysis is shown in Algorithm 3.

The analysis begins by fitting the data points to a polynomial function of degree 0 (as seen in line 4). The coefficient parameters that best fit the specified degree are returned by the fit method. The observed data is fitted to several functions of degree n, where the maximum possible degree is reached when the highest power value of a term is close or equal to 0. A temporal variable *functionCandidate* stores the returned coefficients of each created function, to evaluate its error and store the function with least errors (line 4- 16). After the maximum degree is reached, the value of the variable *chosenDegree* represents the degree of the function that best fits the observed data to a polynomial function.

Algorithm 3 Polynomial-degree-analysis

```
procedure FITTINGPOLYNOMIAL(observedData, varName)
    $degree \leftarrow 0$
    while ($maxDegreeReached == false$) do
        $fitter \leftarrow PolynomialCurveFitter.create(degree)$
        $functionCandidate \leftarrow fitter.fit(observedData)$
        if ($functionCandidate[degree])! = 0$) then
            $error \leftarrow evaluatePolynomial(observedData, functionCandidate, degree)$
            if ($error < errorTemp$) then
                $errorTemp \leftarrow error$
                $chosenDegree \leftarrow degree$
            end if
        else
            $maxDegreeReached \leftarrow true$
        end if
        $degree \leftarrow degree + 1$
    end while
    $fitter \leftarrow PolynomialCurveFitter.create(chosenDegree)$
    $function \leftarrow fitter.fit(observedData)$
    $fittedEquation \leftarrow prepareEquation(function, observedData)$
    return $error$
end procedure
```

5.2 Exponential Fitting

In this section the data to be analyzed is assumed to come from an exponential function. The approach consists of first converting the incoming data to a linear function, in order to retrieve the slope and interception values. Once the parameters are obtained, they are used in order to represent an exponential function with the following structure: $y = ae^{bx}$, where a is the slope of the linear equation and b the interception.

5.2.1 Exponential-Analyzer Algorithm

This implementation utilizes the *SimpleRegression* class from the *oapache-commons-math3* library. It estimates an ordinary least squares regression model with one independent variable.

The algorithm begins by instantiating a new *SimpleRegression* object called sr, that stores the information of the observed data. As the library works with linear equations, the logarithm of each observed value y is added to the data of sr. Once all the data is gathered, the interception is obtained by the method *getIntercept*() and stored in variable a. The slope is retrieved by the method *getSlope*() and stored in variable b.

And finally, all of the parameters of the fitted exponential function are stored in the array $function$.

Algorithm 4 Exponential-analysis

```
1: procedure FITTINGEXPONENTIAL(observedData, varName)
2:     $sr \leftarrow$ new $SimpleRegression()$
3:     for $j \leftarrow 0; j < observedData.size(); j++$ do
4:         $x \leftarrow observedData.get(j).getX()$
5:         $y \leftarrow Math.log(observedData.get(j).getY()))$
6:         $sr.addData(x, y))$
7:     end for
8:     $a \leftarrow sr.getIntercept()$
9:     $b \leftarrow sr.getSlope()$
10:    $function[0] \leftarrow a$
11:    $function[1] \leftarrow b$
12:    $fittedEquation \leftarrow prepareEquation(function, observedData)$
13:    return $error$
14: end procedure
```

5.3 Harmonic Oscillation Fitting

In this section it is assumed that the data comes from a sinusoidal function. The parameters that need to be obtained to construct the expected function are: α , ω and ϕ. The distance between a peek point from the sinusoidal wave to another peek is called the period (ω). The distance from the center of the wave until its peek value is called the amplitude (α). The phase (ϕ) represents how a wave is horizontally shifted from its original position to the right or to the left. Putting all of the previous parameters together, the structure of our assumed wave function is described as the following: $y = \alpha \cdot cos(\omega \cdot x + \phi)$.

5.3.1 Harmonic-Analyzer Algorithm

The $HarmonicFitter$ class is utilized in this implementation from the library *apache-commons-math3*, which specializes on fitting data to sinusoidal functions.

The analysis begins by creating a $fitter$ object of type $HarmonicFitter$, which stores the observed data. All of the information from the *observedData* list is stored in $fitter$ without any modification. After storing the data, the method fit is the one that fits the data to a sinusoidal function and returns an array $function$. This last array contains the amplitude, period and phase parameters that will help to construct the cosine function.

Algorithm 5 Harmonic-analysis

```
1: procedure HARMONICEXPONENTIAL(observedData, varName)
2:     $fitter \leftarrow$ new $HarmonicFitter()$
3:     for $j \leftarrow 0; j < observedData.size(); j++$ do
4:         $fitter.addObservedPoint(observedData.get(j))$
5:     end for
6:     $function \leftarrow fitter.fit()$
7:     $fittedEquation \leftarrow prepareEquation(function, observedData)$
8:     return $fittedEquation$
9: end procedure
```

5.4 Equation Structure

Although all of the fitted functions have different structures and parameters, they are all treated equally as equations with the following attributes: The *name* attribute is used as a unique identifier, *data* stores the mathematical expression of the fitted function, *type* stores the type of function that the equation resembles (e.g. polynomial, exponential or sinusoidal), *initialTime* represents the first time unit when the function is active in the model, whereas *endTime* the last time unit that the function is active in the model.

Remark. The *initialTime* variable is initialized with the value of the first data point from a trace. Nonetheless, the values from both *initialTime* and *endTime* can be modified at run-time in the analysis process.

5.5 Construction of Equations

As seen from the previous algorithms, *prepareEquation* is always called after each analysis is finished. The purpose of this method is to assemble the parameters of a fitted function, and to construct an *Equation* object E that represents the behavior of the current data. It receives a fitted function from an analysis ($function$) and the observed data of the current buffer ($observedData$). The parameters of the $function$ variable are composed into its respective mathematical representation and stored in the *data* field of

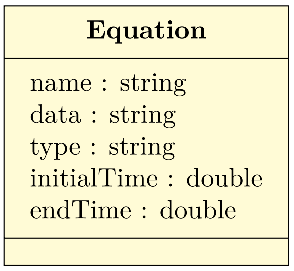


Figure 5.1: Structure of the Equation class

E. The values for *initialTime* and *endTime* are set based on the time values stored in *observedData*.

5.6 Fitted Equations Stack

This structure represents a last-in-first-out (LIFO) stack of *Equation* objects, where only the best fitted equations are stored. The procedure of adding a new equation to the stack is explained in Algorithm 6 (please note that this method is called from Algorithm 2). A *fittedEquation* is automatically added to the stack whenever it is empty. Otherwise, a reference of the last equation of the stack is taken by the *peek* function. If the *data* attribute of the last equation is the same as the one from *fittedEquation*, then both equations have the same function. In this case, only the upper bound of the last equation is replaced with the upper bound of *fittedEquation* (no equation is added to the stack). But if an equation with a new function is found, then it is automatically added to the stack.

Algorithm 6 Store-Equation

```
1: procedure STOREEQUATION(fittedEquation)
2:     if ($stack.isEmpty()$) then
3:         $stack.push(fittedEquation)$
4:     else
5:         $lastFittedEquation \leftarrow stack.peek()$
6:         if ($lastFittedEquation.getData() == fittedEquation.getData()$) then
7:             $lastFittedEquation.updateUpperBound()$
8:         else
9:             $stack.push(fittedEquation)$
10:        end if
11:    end if
12: end procedure
```

Another useful feature of the stack is that it allows to determine whether the new incoming data of the buffer is correctly fitted by the previous fitted equation or not. This procedure is done in the first lines of Algorithm 2. It consists of solving the last fitted function with the new incoming time unit of the buffer (after it slided). If the evaluated value is close to the original value of the trace, then the new data is fitted by the previous equation. Whenever this happens, only the time duration of the previous equation is updated based on the times that the buffer slided. But if the difference of the evaluated value and the original value of the trace surpasses a specific threshold, it means that the previous function is not able to fit the new incoming data with the same function, then a new analysis is performed.

6 UPPAAL Model Scheme Design

We can now state that a set of equations describe the behavior of a set of traces. A behavior is referred as the functionality of a location from a timed automaton, which is triggered by the execution of an edge at a certain time of a simulation. This chapter will explain how a set of equations can be assembled in order to represent a probabilistic model scheme in UPPAAL.

6.1 Templates

It is the instance of a process in UPPAAL that represents a timed automaton as a model, and that resembles the functionality of a system. As an *Equation* object has a function on its *data* field that describes the behavior of a variable, a template is created for every different variable that is identified in a stack of equations.

6.2 Locations

It is the representation of the state of a model or system, which is capable of executing a functionality. Such functionality is the reference of the stored expression in the *data* field of an equation object. Every location created in a model is based on a fitted equations that was stored in the stack of an analysis.

6.3 Time in UPPAAL

Taking into account that a clock variable in UPPAAL represents the time units of the simulation of a model. Two clocks are used in each created model to synchronize the functionalities of every location from each template. One clock with the name of *tg* is declared globally for every constructed model. It references the time that passes by in a simulation of a model and it is referable from any declared template. A second clock is declared locally in each template with the name of *ti*, which is only visible in the template that it was declared. It helps to coordinate the actions of the edges from a location.

6.3.1 Guards

A guard is used in this implementation to synchronize the actions of every location. It ensures that a functionality in a location is executed only after 1 time unit has passed in the simulation, without surpassing the time given by the *endTime* of the equation that the location resembles.

6.3.2 Invariants

An invariant is used in this implementation to represents the time units that a location is active in a simulation. It ensures that a location performs its functionality once 1 time unit passes by in a simulation and only until tg reaches the $endTime$ of the location. A location is forced to take an edge to another location whenever the $endTime$ is reached.

6.4 Edges

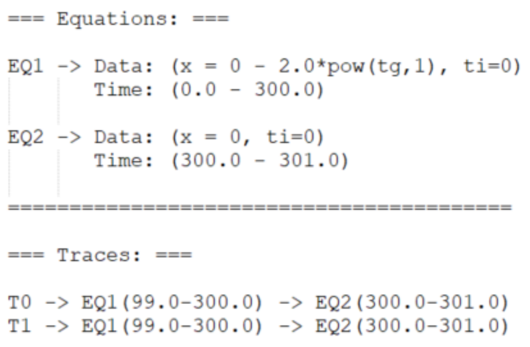
A location in this implementation always has a self-pointing edge, in which a guard controls the time of execution of a functionality. When an edge of a location points to another location, a guard is used in the edge to restricts the time in which the location is allowed to travel to its destination. The clock constraint that has to be satisfied in the previously mentioned edge is expressed as: $tg \geq endTime$.

6.5 Branch Points Functionality

The utility of branch points in this implementation is crucial for creating probabilistic models, as they coordinate the flow of the outgoing edges of a location. They decide which edge a location should take. Every template possesses an initial urgent location with an edge that goes to an initial branch point. This allows to treat a template as a fully connected graph, where a location is a node that can be attached or detached from any position of the graph.

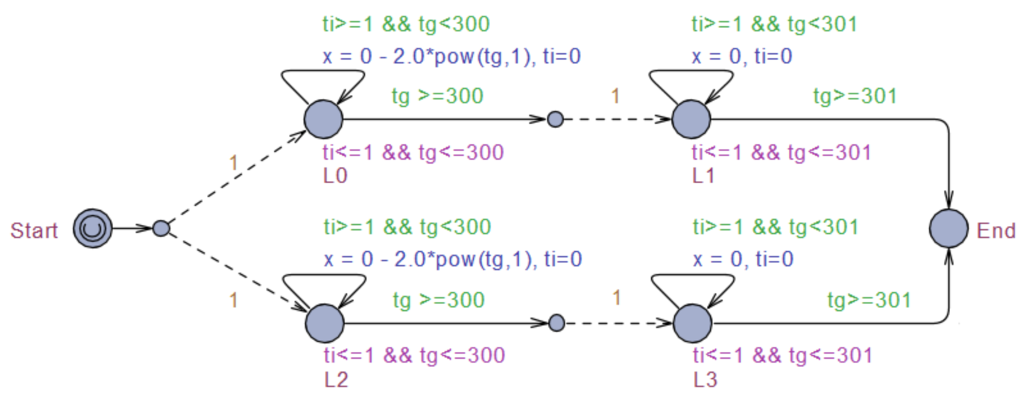
6.6 Probabilistic Scheme Example

Figure 6.1: Example of the representation of data from traces as equations



```
=== Equations: ===

EQ1 -> Data: (x = 0 - 2.0*pow(tg,1), ti=0)
       Time: (0.0 - 300.0)

EQ2 -> Data: (x = 0, ti=0)
       Time: (300.0 - 301.0)

========================================

=== Traces: ===

T0 -> EQ1(99.0-300.0) -> EQ2(300.0-301.0)
T1 -> EQ1(99.0-300.0) -> EQ2(300.0-301.0)
```

Figure 6.2: Example of the structure of the models to be recreated



Every trace contains a stack of fitted equations, which resemble the behavior of a variable from a simulation. $T0$ and $T1$ have the same behavior, described by function $EQ1$ that is active from time 99 to 300, and $EQ2$ that is active from time 300 - 301. If we take a look at location $L0$, it is only active until time unit 300 of the simulation (referenced by tg) is reached. Its functionality is based on $EQ1$, which decrements the value of x by 2 and reset the value of ti. This functionality is controlled by the clock constraints of ti and tg, which only allows a location to perform its functionality once 1 time unit passes by in the simulation, until it reaches is upper bound time limit (referenced by tg). Every equation from a trace is transformed to a location and attached to the model starting from the first location named *Start*, until the last one called *End*. This representation formalizes the design of the scheme of our probabilistic model. The implementation of the integration of edges between locations is explained in Chapter 7.

7 UPPAAL Probabilistic Model Implementation

A location may have many outgoing edges, in which only 1 edge must be taken at a time. This feature is only possible with the use of branch points, that coordinate all outgoing edges of a location based on a probabilistic value. In this chapter we will explain how a probabilistic model scheme is constructed, from the integration of the fitted equations obtained from all analyses.

7.1 Final Analysis Processing

After completing the analyses of each trace obtained from a model simulation, the results are gathered in a list of *analysis* objects. The way that the results are handled is described in Algorithm 7. Given a stack of equations obtained from the function fitting algorithms, a new template is created for each different variable that is found. If a template for a variable was already created, this last one is only updated. Each equation retrieved from the stack is represented as a location, that is integrated in the final probabilistic model. The procedure of how branch points, edges and locations are connected to each other is explained in the next sections.

Algorithm 7 Final-Analysis

```
1: procedure FINALANALYSIS(analysisArray)
2:     for $i \leftarrow 1, analysisArray.Size()$ do
3:         $analysis \leftarrow analysisArray[i]$
4:         $template \leftarrow analysis.getTemplateOrCreate()$
5:         $result \leftarrow analysis.getStack()$
6:         $template.joinLocations(result)$
7:     end for
8: end procedure
```

7.2 Creating Locations

The *equationStack* parameter of Algorithm 8, contains all of the fitted equations of a trace. A new location with an invariant and a self-pointing edge is created, based on the information of each equation obtained from the stack. Before connecting the new location to a component of the template, we first reference a branch point (variable *branchPoint*) that is already integrated in the template. As previously mentioned, every template is initialized with an initial-urgent location called *Start*, that has an outgoing edge leading to the initial branch point called *BP*. The variable *branchPoint* always references *BP* every time the first equation from a stack is retrieved. Otherwise, *branchPoint* references the outgoing branch point of the previous location that was attached to the model. In case that a previous location has no outgoing branch point, then a new one is created and attached to it. The *previousLocation* variable always refers to the last location that was added to the template. The method *probabilityModelling* is the one that actually incorporates a new location to the template and returns a reference of it. In order to keep consistency and efficiency of the constructed probabilistic model, 5 cases must be considered before adding a new location to a template. The explanation of the *probabilityModelling* method is shown on the next section.

Algorithm 8 Join-locations

```
1: procedure joinLocations(equationStack)
2:     $i \leftarrow 0$
3:     for $FittedEquation\ equation : st$ do
4:         $location \leftarrow addNewLocation(equation)$
5:         if $i == 0$ then
6:             $branchPoint \leftarrow getInitialBP()$
7:         else
8:             $branchPoint \leftarrow previousLocation.getOutgoingBP()$
9:             if $branchPoint == null$ then
10:                $previousLocation.createNewBranchPoint()$
11:            end if
12:        end if
13:        $previousLocation \leftarrow probabilityModelling(branchPoint, location, i)$
14:        $i \leftarrow i + 1$
15:    end for
16: end procedure
```

7.3 Probabilistic Modelling

The *probabilityModelling* method consists of first verifying the functionality of the new location that is going to be added to the template, before attaching it to any branch point (as seen in Algorithm 9). We begin by first performing an inspection of the outgoing edges that the current branch point (*branchPoint*) already possesses (if any). The *equation* variable references the equation of an outgoing location (*location*) of *branchPoint*, that is already incorporated in the template. The *newEquation* variable references the equation of the new location (*newLocation*) that is going to be incorporated in template. If the fitted functions (or *data* attributes) from *equation* and *newEquation* are equal, it means that there exists a location in the template with the same functionality as the new location to be incorporated. Whenever this happens, we proceed and verify the time duration of both equations. The *newLocation* might be added to the model, depending on the difference of time duration between the two equations. There are five possible scenarios that can occur when analyzing the equations of a trace in method *processEquationData*, which are explained in Sections **??** - 7.3.5 (Notice that *EQ* is the equation of *location* and *EQ′* the one from *newLoc*). Whenever *branchPoint* has no edges pointing to other locations with the same functionality as *newLocation*, then a new outgoing edge is added to *branchPoint* that leads to *newLocation*. A probabilistic model is created after Algorithm 9 terminates, which represents the recreated model that describes the trending of the data from the traces that were analyzed. The verification of the similarities and differences between the recreated model and the original one are discussed in the next chapter.

Algorithm 9 Probability-Modelling

```
1: procedure ProbabilityModelling(branchPoint, newLocation, i)
2:     $newEquation \leftarrow newLocation.getEquation()$
3:     for ($outGoingEdge : branchPoint.getOutgoingEdges()$) do
4:         $location \leftarrow outGoingEdge.getLocation()$
5:         $equation \leftarrow location.getEquation()$
6:         if $equation.getData() == newEquation.getData()$ then
7:             $previousLoc \leftarrow processEquationData(newEquation, equation)$
8:             return $previousLoc$
9:         end if
10:    end for
11:    $addProbEdge(branchPoint, newLoc)$
12:    $previousLoc \leftarrow newLoc$
13:    return $previousLoc$
14: end procedure
```

7.3.1 Case 1 (Identical Equations)

If both of the equations start and end at the same time, it means that both of them are identical. In this case *newLoc* is deleted from the template and the edge that leads to *location* is incremented by 1. The returned location is *location*.

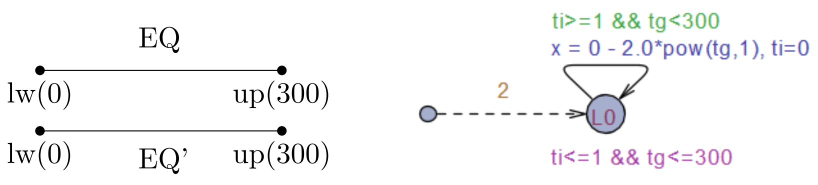


Figure 7.1: Example of the addition of the same equation to a model

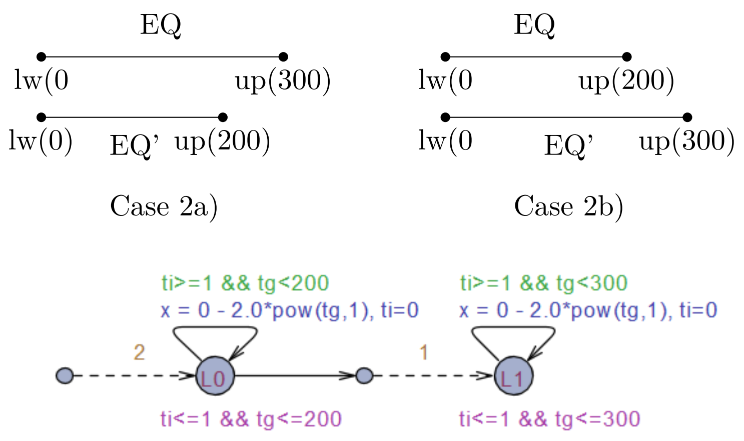


Figure 7.2: Example of the addition of an equation that begins at the same time to a model

7.3.2 Case 2 (Same Starting-Time Equations)

Both equations start at the same time, but end at different ones. Location *newLoc* is added to the model with a slight modification of its time bounds. Its lower bound remains the same, but the upper bound is updated to the smallest *upperBound* value from the two equations EQ and EQ'. The location returned for both cases 2a) and case 2b) is *newLoc*.

7.3.3 Case 3 (Same Ending-Time Equations)

Both equations begin at different times, but finish at the same one. The new location *newLoc* is introduced in the model with a slight modification of its time bounds. Its lower bound is updated to the *lowerBound* from the equation that has more time duration among EQ and EQ'. The upper bound is updated to the *lowerBound* of the equation with the smallest duration time. Once *newLoc* is integrated in the model, *location* is updated by setting its *lowerBound* to the *upperBound* of *newLoc*. The returning location for both cases 3a) and 3b) is always *newLoc*.

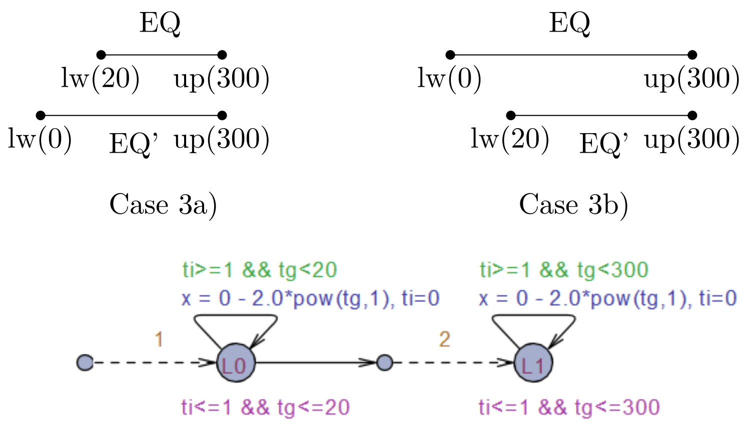


Figure 7.3: Example of the addition of an equation that ends at the same time to a model

7.3.4 Case 4 (Contained Equations)

In this scenario, an equation has a time duration that is a sub-interval of the time duration of the other equation. The new location *newLoc* is introduced in the model with a slight modification of its time bounds. Its lower bound is updated to the *lowerBound* from the equation that highest duration time. The upper bound is updated to the *lowerBound* of the equation with the lowest duration time. Once *newLoc* is integrated to the model, it now requires to 2 outgoing edges. The first outgoing edge must go to a location with the *upperBound* of EQ, while the other must go to another location with the *upperBound* of EQ'. The returning location for both cases 4a) is *newLocation*.

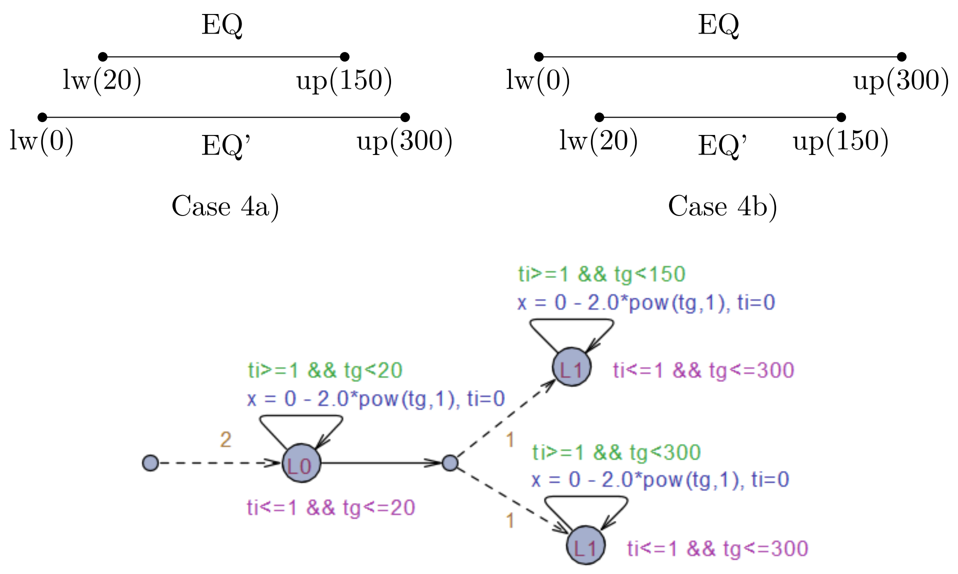


Figure 7.4: Example of the addition of an equation with a sub-interval of time to a model

7.3.5 Case 5 (Continued Equations)

Here an equation begins right after the previous one finished. The new location *newLoc* is only added as an outgoing edge of the location that was already in the model (*location*). The returning location in this case is always *newLoc*.

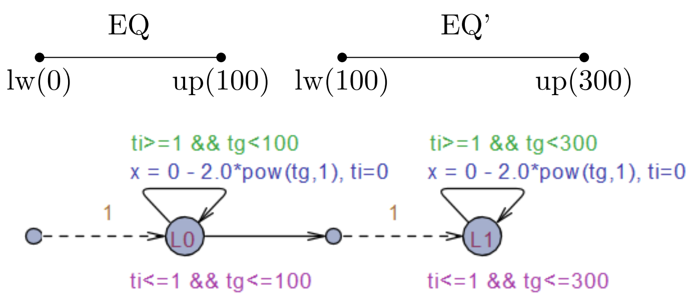


Figure 7.5: Example of the addition of an equation that starts at the ending time of another equation to a model

8 Model Comparison

As the source of the traces to be analyzed come from the execution of a query in UPPAAL, we us the same query to also obtain traces from the recreated model. This ensures that the same number of traces are extracted from the two models and under the same conditions. It is explained in the following section how the comparison of the traces from the two models is performed, and how the compatibility of their functionalities is determined.

8.1 The Trace-Comparison Algorithm

For simplicity, we assume that the query was already executed in the recreated model and that the traces were stored in the variable *simResults*. We also assume that the original trace from the first simulation is already stored in list of simulations called *originalResults*. The filtering process of the data from the traces of a recreated model is the same as the one explained in Chapter 4. The only difference is that when a numerical trace is found in *simResults*, we now proceed and compare its values against the traces from *originalResults*.

Algorithm 10 Trace-Comparison

```
procedure traceComparison(simResults)
    for $i \leftarrow 1, simResults.Size()$ do
        $trace \leftarrow simResults[i]$
        for $j \leftarrow 1, trace.Size()$ do
            $traceData \leftarrow trace[j]$
            for $k \leftarrow 1, originalResults.Size()$ do
                $originalTrace \leftarrow originalResults[k]$
                for $l \leftarrow 1, originalTrace.Size()$ do
                    $originalTraceData \leftarrow originalTrace[l]$
                    $error \leftarrow compareTraces(originalTraceData, traceData)$
                end for
            end for
        end for
    end for
end procedure
```

The comparison of the data of a trace from a recreated model consists on searching through all of the original traces and to compare the values of a variable at all time units of the simulation. As all traces have a tree map structure, it is very simple to retrieve the value of a variable at an exact time unit, by using the $getValue()$ method. It only requires a key that represents the x value of a trace (or time unit of the simulation) in order to return the value of the variable that was analyzed. The $getValue()$ method is used to obtain a value of a variable from *simResults*, based on the key of every trace stored in *originalResults*. If a value from *simResults* is obtained, then the original value from *originalResults* is retrieved by using the same key. Whenever the two values differ significantly, it means that a functionality of the recreated model does not match the functionality of the original model in a certain time unit. And if a value cannot be retrieved from *simResults*, it means that the recreated model was not able to identify a behavior from a certain time unit of the original model. This approach may give negative results when comparing traces of non-deterministic systems. It might be the case that two models output different traces with the same non-deterministic functionality. In this scenario, it is most likely that two models are distinguished as incompatible even if they are not.

9 Case Study

This chapter presents two experiments that demonstrate how the data from traces can be transformed to probabilistic models. The first experiment consists of recreating a model by analyzing its traces. The second experiment consists of representing a probabilistic model by only analyzing raw data from a file.

9.1 Experiment 1

First, we explain the functionalities of the original model to be recreated in Section 9.1.1. Then, we extract a set of traces from the previous model and explain the interpretation of the data in Section 9.1.2. Afterwards, we show the fitted equations of the traces after the analyses were performed in Section 9.1.3. And we finalize by explaining the functionality of the reconstructed model in Section 9.1.4.

9.1.1 Original Model Functionality

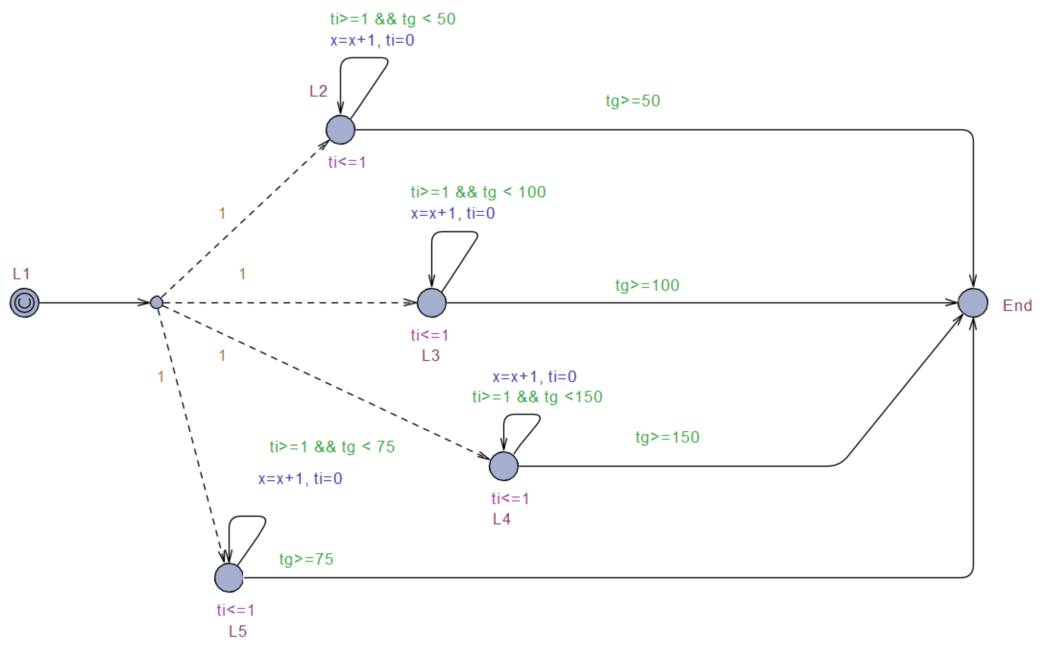
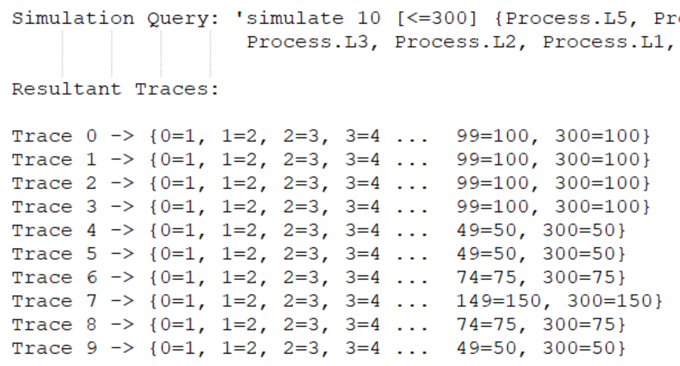


Figure 9.1: Original model from which traces were analyzed

The functionality of the model to be reconstructed is to increment the value of variable x on different time intervals. The starting location $L1$ has an outgoing edge pointing to a branch point. This last one has several outgoing edges that lead to different locations, where each location executes the same functionality as $L1$ in different time unit ranges (e.g. 0-50, 0-100, 0-150, 0-75). The probability of the outgoing edges from the first branch point have the same value, which simulated a non-deterministic behavior for the edge that $L1$ can execute in every simulation.

9.1.2 Trace Extraction

The query used to extract the traces from the model is depicted in the figure below. It performs 10 simulation rounds with a duration of 300 time units and retrieves the traces from all the locations of the model.



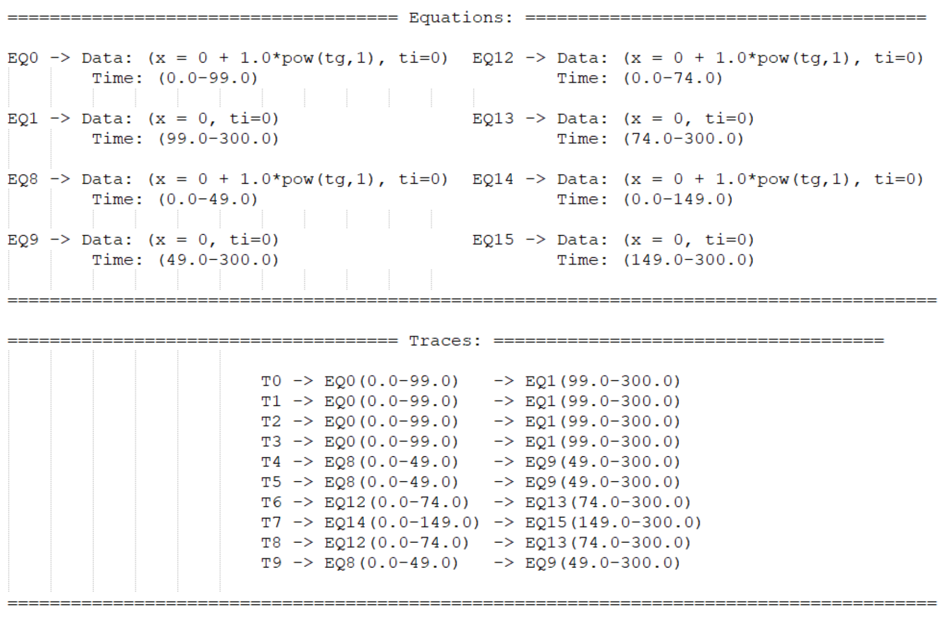
```
Simulation Query: 'simulate 10 [<=300] {Process.L5, Process.End, Process.L4,
                   Process.L3, Process.L2, Process.L1, tg, Process.ti, Process.x}'

Resultant Traces:

Trace 0 -> {0=1, 1=2, 2=3, 3=4 ...  99=100, 300=100}
Trace 1 -> {0=1, 1=2, 2=3, 3=4 ...  99=100, 300=100}
Trace 2 -> {0=1, 1=2, 2=3, 3=4 ...  99=100, 300=100}
Trace 3 -> {0=1, 1=2, 2=3, 3=4 ...  99=100, 300=100}
Trace 4 -> {0=1, 1=2, 2=3, 3=4 ...  49=50, 300=50}
Trace 5 -> {0=1, 1=2, 2=3, 3=4 ...  49=50, 300=50}
Trace 6 -> {0=1, 1=2, 2=3, 3=4 ...  74=75, 300=75}
Trace 7 -> {0=1, 1=2, 2=3, 3=4 ...  149=150, 300=150}
Trace 8 -> {0=1, 1=2, 2=3, 3=4 ...  74=75, 300=75}
Trace 9 -> {0=1, 1=2, 2=3, 3=4 ...  49=50, 300=50}
```

Figure 9.2: Example of the extraction of the traces from a model

The value of x is incremented by 1 from the beginning of the simulation (at time unit 0) for each time unit that passes by in the simulation. It is the case for all traces that the value of x ceases to increment and remains constant after a time unit is reached, until the end of the simulation (when time unit 300 is reached).

9.1.3 Analysis Interpretation

We only discuss in this section the results of the function fitting algorithms from Chapter 5. The equations that were constructed based on the analysis of the traces can be seen on the next figure. All of the shown functions were constructed with the syntax of UPPAAL. As expected, the incrementing phase of x was expressed for all traces as a polynomial function of degree one: $x = 0 + 1 * pow(tg, 1)$. The procedure of resetting the value of x was expressed as a polynomial function of degree 0: $x = 0$. And finally, a new set of traces was composed by gathering all of the fitted equations from each trace.



```
==================================== Equations: ====================================

EQ0 -> Data: (x = 0 + 1.0*pow(tg,1), ti=0)  EQ12 -> Data: (x = 0 + 1.0*pow(tg,1), ti=0)
        Time: (0.0-99.0)                             Time: (0.0-74.0)

EQ1 -> Data: (x = 0, ti=0)                  EQ13 -> Data: (x = 0, ti=0)
        Time: (99.0-300.0)                           Time: (74.0-300.0)

EQ8 -> Data: (x = 0 + 1.0*pow(tg,1), ti=0)  EQ14 -> Data: (x = 0 + 1.0*pow(tg,1), ti=0)
        Time: (0.0-49.0)                             Time: (0.0-149.0)

EQ9 -> Data: (x = 0, ti=0)                  EQ15 -> Data: (x = 0, ti=0)
        Time: (49.0-300.0)                           Time: (149.0-300.0)
=========================================================================================

==================================== Traces: ====================================

                        T0 -> EQ0(0.0-99.0)    -> EQ1(99.0-300.0)
                        T1 -> EQ0(0.0-99.0)    -> EQ1(99.0-300.0)
                        T2 -> EQ0(0.0-99.0)    -> EQ1(99.0-300.0)
                        T3 -> EQ0(0.0-99.0)    -> EQ1(99.0-300.0)
                        T4 -> EQ8(0.0-49.0)    -> EQ9(49.0-300.0)
                        T5 -> EQ8(0.0-49.0)    -> EQ9(49.0-300.0)
                        T6 -> EQ12(0.0-74.0)   -> EQ13(74.0-300.0)
                        T7 -> EQ14(0.0-149.0)  -> EQ15(149.0-300.0)
                        T8 -> EQ12(0.0-74.0)   -> EQ13(74.0-300.0)
                        T9 -> EQ8(0.0-49.0)    -> EQ9(49.0-300.0)

=========================================================================================
```

Figure 9.3: Original model analysis result from Experiment 1

9.1.4 Recreated Model

A new mode was constructed based on the traces of fitted equations from the previous section. Each location executes one of the mentioned polynomial functions and also resets the value of the clock ti, which ensures that the equations are only triggered every time unit. This model represents the exact behavior of the extracted traces obtained from the original model. Given the starting and ending times from each fitted equation, it was concluded that all equations are active at time 0 and increment the value of x by 1 until time unit 49 is reached (as seen in location $L16$). A branch point was introduced to coordinate the path that location $L16$ should take after the time of the simulation (referenced by tg) surpasses 49 units. Given the weighted probabilities of the outgoing edges (e.g. 4,3,2,1), the most probable edge to be taken by $L16$ is the one leading to $L0$, followed by the one from $L18$, $L24$ and $L28$. The recreated model could not resemble the exact behavior of the original model because the traces that were analyzed do not reflect the same non-deterministic behavior.

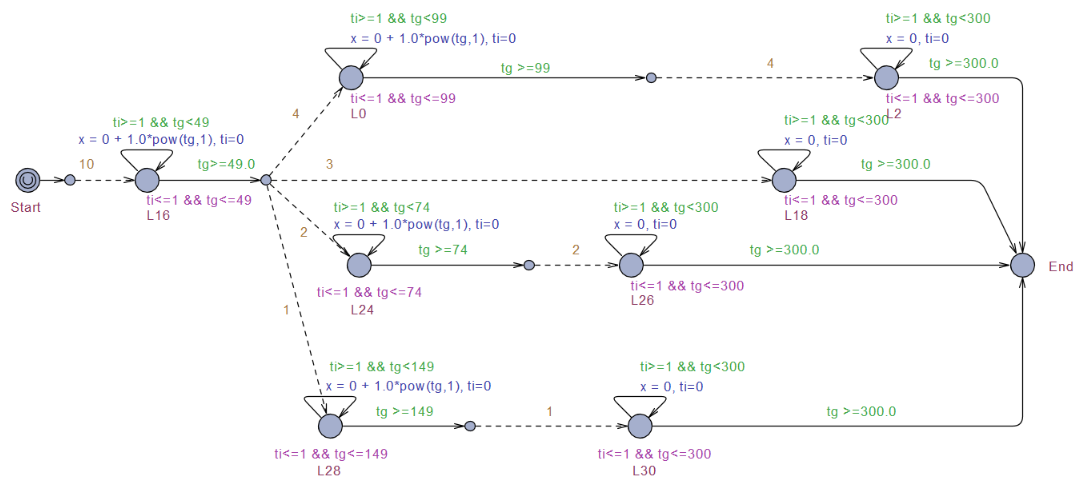


Figure 9.4: Recreated model version of the original model used in Experiment 1

9.2 Experiment 2

The motivation of this experiment is to analyze data with a behavior that is similar to the sinusoidal wave of a heart beat. We start by showing a visual representation of the data to be analyzed and discuss its functionality in Section 9.2.1. Then we explain the different fitted equations that can be obtained depending on the size of the *buffer* used by an analyses in Section 9.2.2. Afterwards we show the results of some experiments from Section 9.2.3 to 9.2.6. And finally we interpret the functionality of the best recreated model section 9.2.7.

9.2.1 Analysis Of The Data

The data to be analyzed comes as a set of x and y values from a file that is visualized as a sine wave, depicted in the figure below.

The amplitude α of the wave has a value of 1, the frequency ω a value of $2\pi * t$ (t is a time unit of the simulation), and the phase shift ϕ a value of 0.

9.2.2 Analysis Approach

The objective of this section is to demonstrate how data can be differently modelled, depending on the perspective from which it is analyzed. Each of the next analyses was performed with different *buffer* sizes and a fixed time step value of 1. The retrieved raw data is treated as a trace, in order to conduct all of the analyses.

9.2.3 Analysis (Buffer Size 15)

If the *buffer* is initialized with a size of 15, then all of the data is fitted to a set of polynomial function of degree 3 and 4.

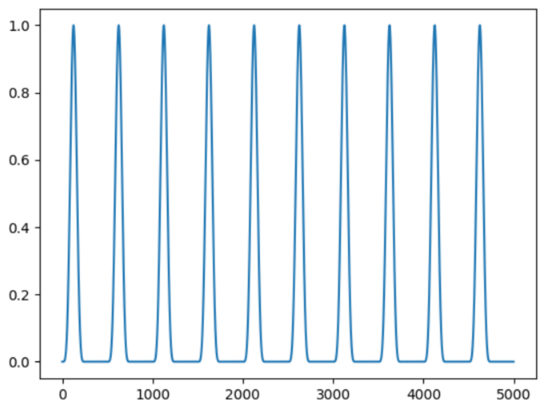


Figure 9.5: Plot of the data from Experiment 2

When the wave ascends from value 0 to 1, it is described as:

$x = 0.00000026 + -0.00000105 * pow(tloc, 1) + 0.00000061 * pow(tloc, 2) + -0.00000012 * pow(tloc, 3) + 0.00000001 * pow(tloc, 4)$

When the wave starts descending from value 1 to 0, it is described as a combination of the functions:

$x = 0.05220597 + -0.00869902 * pow(tloc, 1) + 0.0004698 * pow(tloc, 2) + -0.00000288 * pow(tloc, 3)$
$x = 0.05153081 + -0.00750662 * pow(tloc, 1) + 0.00052926 * pow(tloc, 2) + -0.00000413 * pow(tloc, 3)$

The plotted data from the previous functions resembles the one from the original data

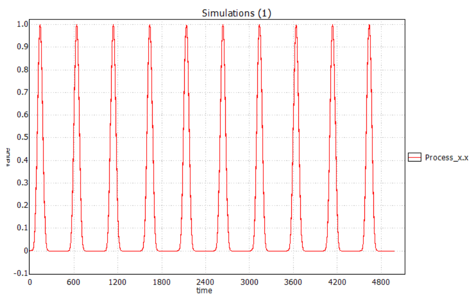


Figure 9.6: Recreated sine wave trace (Buffer Size 15) from experiment 2

because only a small fraction of the data was analyzed at a time and it could always be fitted to a polynomial function.

9.2.4 Analysis (Buffer Size 125 and 250)

When the *buffer* is initialized with a size of 125, the data is first fitted to a function of degree 4 that is unable to detect the peek of the sine wave:

$$x = -0.01705251 + 0.00482419 * pow(tloc, 1) + -0.00029337 * pow(tloc, 2) + 0.00000598 * pow(tloc, 3) + -0.00000003 * pow(tloc, 4)$$

Once a peek is detected, the parameters of the previous function are re-adjusted to the following set of functions (similar as the previous experiment):

$$x = 0.05220597 + -0.00869902 * pow(tloc, 1) + 0.0004698 * pow(tloc, 2) + -0.00000288 * pow(tloc, 3)$$

$$x = 0.05153081 + -0.00750662 * pow(tloc, 1) + 0.00052926 * pow(tloc, 2) + -0.00000413 * pow(tloc, 3) + 0.00000001 * pow(tloc, 4)$$

The only inconvenient is that at some points of the simulation (e.g. between time 500 and 600), another function is detected and causes noise to the simulation:

$$x = 0.00008034 * cos(0.0697996 * tloc + 4.02811424)$$

A similar behavior is seen when the *buffer* is initialized with a size of 250. The structure of the fitted functions remains the same, with the only difference of the size of their coefficients.

Parameter of the new $FittedFunctions$:

$$x = 0.19249276 + -0.02657333 * pow(tloc, 1) + 0.00070873 * pow(tloc, 2) + -0.00000482 * pow(tloc, 3) + 0.00000001 * pow(tloc, 4)$$

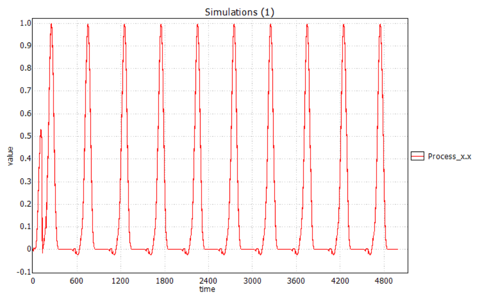


Figure 9.7: Recreated sine wave trace (Buffer Size 125) from experiment 2

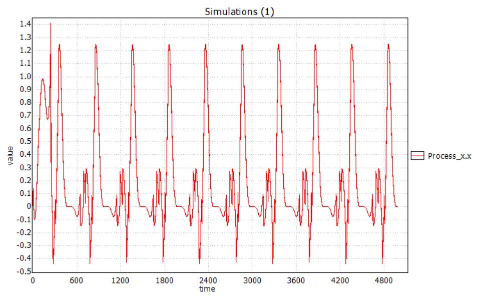


Figure 9.8: Recreated sine wave trace (Buffer Size 250) from experiment 2

$x = -0.13212867 + 0.04755968 * pow(tloc, 1) + -0.00067739 * pow(tloc, 2) + 0.00000322 * pow(tloc, 3) + -0.00000001 * pow(tloc, 4)$

$x = -0.20092568 + 0.03322046 * pow(tloc, 1) + -0.00031573 * pow(tloc, 2) + 0.00000076 * pow(tloc, 3)$

$x = 0.00139367 * cos(0.03486086 * tloc + 6.16261857)$

Due to the fact that more data is being analyzed, the *fitter* cannot correctly identify the properties of the sine wave.

9.2.5 Analysis (Buffer Size 625)

When the *buffer* is initialized with a size of 625, the beginning of the data is identified as a third-degree polynomial function:

$x = -0.0095713 + 0.00787921 * pow(tloc, 1) + -0.00003912 * pow(tloc, 2) + 0.00000005 * pow(tloc, 3), ti = 0)Time : (0.0 - 624.0)$

After some time of the simulation, the fitter is able to identify when the sine wave ascends to the peek by two different functions:

$x = 1.0 * pow(cos(0.01256637 * tloc + 4.73752172), 5), x = (x < 0.0)?0.0 : x, ti = 0)$
$x = 0.31975163 * (pow(2.718281828459045, -0.01528879 * tloc))$

But a problem arises when the sine wave descends to the value of 0 because the fitted function changes to a cosine function with a negative peek:

$x = 0.33553271 * cos(0.00703253 * tloc + 3.99344951), ti = 0)$

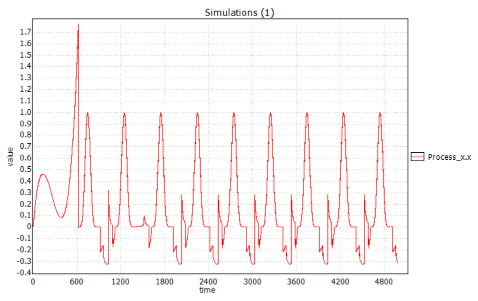


Figure 9.9: Recreated sine wave trace (Buffer Size 625) from experiment 2

9.2.6 Analysis (Buffer Size 750)

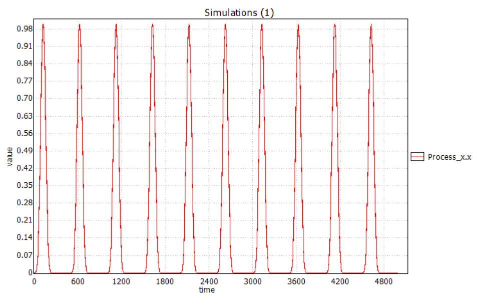
When the *buffer* is initialized with a size of 750, only one equations is fitted.

$FittedEquation$:

$x = 1.0 * pow(cos(0.01256637 * tloc + 4.71238898), 5)$

This is only possible because the amount of data is large enough for the analysis to identify two consecutive peaks of the sine wave. As the sine wave has the same behavior through all of the simulation, one function is able to describe its tendency.

Figure 9.10: Sine Wave Trace Buffer Size 750 Experiment 2



9.2.7 Recreated Model

The best found scenario was of course when the data was analyzed with a *buffer* size of 750. Only one location that executes an exponential cosine functions is created in the recreated model, which is enough to reproduce the exact same functionality of the original data.

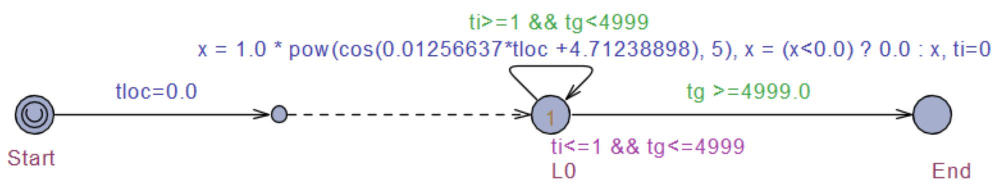

ti>=1 && tg<4999
x = 1.0 * pow(cos(0.01256637*tloc +4.71238898), 5), x = (x<0.0) ? 0.0 : x, ti=0
tloc=0.0
tg >=4999.0
1
ti<=1 && tg<=4999
Start
L0
End

Figure 9.11: Sine Wave Recreated Model Experiment 2

10 Future Work

The implementation of this project is able to recreate the traces from a model. The only inconvenience is that the configuration of the analyses must be correctly setup to obtain good results. We will discuss in this section the pros and cons of the implemented tool of this project.

The main advantage is that a solid structure for analyzing the data of traces has been sucessfully established. It can be extended for analyzing more complex types of functions (e.g. exponential sinusoid functions). In addition, the structure of the *buffer* and its window shift process can be easily modified to improve the precision of the analysis.

The main disadvantage is that the size and *timeStep* value of a *buffer* must be set prior to the execution of an analysis, and that they remain unmodified through all the analysis process. One cannot guarantee efficiency in the recreated models with this configuration. A model might be successfully recreated with the expected functionalities, but with an inefficient use of locations and edges. If a model was recreated with redundant locations, it means that the analysis of the traces fitted redundant functions mainly because of the configuration of the *buffer*, or due to the error tolerance that the analysis uses.

The efficiency of the recreated models can be improved by integrating an optimization phase that iterates the structure of the probabilistic model, with the aim of eliminating redundant locations. Another improvement might be to integrate an initial *on-line* phase, in which the algorithm learns the most suitable configuration setup for a trace before analyzing it.

11 Conclusion

This thesis was focused on identifying and analyzing the parameters of the traces of a system, in order to reconstruct its functionality as a probabilistic model. The aim was to fit the behavior of each obtained trace into a set of mathematical equations. The recreation of a model was elaborated by verifying and incorporating the data of each fitted equation in a probabilistic model with UPPAAL. Once the recreated version from an original model was completed, the compatibility of their functionalities was identified by comparing the data of their traces. We have demonstrated that it is possible to recreate the behavior of a model based on the analysis of its traces. But we also discovered how the complexity of an analysis rises when it comes to recreate the traces from non-deterministic models. Our approach strictly depends on the amount of data that can be retrieved from a model, and of how it is analyzed in terms of quantity and frequency. We are only able to recreate the functionality from the trace of a system, and to verify whether the functionality of the recreated model matches the one from the system from which the traces were obtained. If we were to fully recreate the functionality of a system based on its traces, we would need to set up an environment in which a system constantly feeds information to the other one before performing the analyses. This would enable the ability of identifying non-determinstic behaviors and to also optimize the recreated model in an *on-line* fashion.

Bibliography

[1] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[2] Samuel Eilenberg. *Automata, languages, and machines.* Academic press, 1974.

[3] Per Christian Hansen, Victor Pereyra, and Godela Scherer. *Least squares data fitting with applications.* JHU Press, 2013.

[4] Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: a redundancy-free approach to active automata learning. In *International Conference on Runtime Verification*, pages 307–322. Springer, 2014.

[5] Alexander Maier. Online passive learning of timed automata for cyber-physical production systems. In *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*, pages 60–66. IEEE, 2014.

[6] Jorge Nocedal and Stephen J Wright. Least-squares problems. *Numerical optimization*, pages 245–269, 2006.

[7] Marcello Pagano and Kimberlee Gauvreau. *Principles of biostatistics.* Chapman and Hall/CRC, 2018.

[8] Frits Vaandrager. A first introduction to uppaal. *Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook*, 18, 2011.