EVENT-BASED AUTOMATON MODEL FOR IDENTIFICATION OF DISCRETE-EVENT SYSTEMS WITH THE AIM OF FAULT DETECTION

Thiago Henrique de Marreiros Cordeiro Machado

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Marcos Vicente de Brito Moreira

Rio de Janeiro
Junho de 2022

EVENT-BASED AUTOMATON MODEL FOR IDENTIFICATION OF DISCRETE-EVENT SYSTEMS WITH THE AIM OF FAULT DETECTION

Thiago Henrique de Marreiros Cordeiro Machado

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Orientador: Marcos Vicente de Brito Moreira

Aprovada por: Prof. Marcos Vicente de Brito Moreira
Prof. Antonio Eduardo Carrilho da Cunha
Prof. Felipe Gomes de Oliveira Cabral
Prof. Gustavo da Silva Viana

RIO DE JANEIRO, RJ – BRASIL
JUNHO DE 2022

Machado, Thiago Henrique de Marreiros Cordeiro

Event-Based Automaton Model for Identification of Discrete-Event Systems with the Aim of Fault Detection/Thiago Henrique de Marreiros Cordeiro Machado. – Rio de Janeiro: UFRJ/COPPE, 2022.

XV, 63 p.: il.; 29, 7 cm.

Orientador: Marcos Vicente de Brito Moreira

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2022.

Referências Bibliográficas: p. 59 – 61.

1. Fault diagnosis. 2. System identification. 3. Discrete-event systems. I. Moreira, Marcos Vicente de Brito. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Dedico este trabalho à minha família por todo o suporte fornecido.

Acknowledgments

Agradeço, em primeiro lugar, à minha mãe Diana Maria de Marreiros Cordeiro (*In Memoriam*) por tudo que fez e deixou de fazer. Infelizmente, não conseguirá me ver concluir mais esta etapa, mas tenho a certeza de que estaria super orgulhosa e feliz do caminho que tenho traçado. Sempre penso em você!

Ao meu avô Geovah Ubirajara Amaral Machado por todo o suporte. As conversas diárias, as perguntas (muitas das vezes repetidas), a preocupação com as minhas decisões e o meu futuro, o suporte total que me deixou numa situação onde pude sempre focar nos estudos, enfim, tudo o que foi e vem sendo feito são uma parte fundamental de toda a minha trajetória! Sou muito feliz e agradecido!

À minha avó Darci Michel Cardoso Machado que sempre tornou a minha vida mais agradável em casa. As conversas no café da manhã e no lanche da tarde sempre dão aquela energia pro dia. Sempre prezando pela organização da casa (o que é uma tarefa difícil aqui), cuidando da rotina da mesma e fazendo almoços e quitutes querendo agradar a todos (o que é impossível). É um trabalho importante e árduo que as pessoas só se dão conta do valor, em geral, quando não é feito.

Ao meu pai Sergio Alexandre Cardoso Machado que faz de tudo para estar sempre presente e tentar me ajudar no que for possível. As discussões, os almoços, os lanches de domingo, os passeios (ou missões?) de bicicleta, enfim, todos os momentos juntos são sempre importantes.

Aos meus irmãos Alexandre Matheus Cordeiro Machado e Raphael de Marreiros Cordeiro Machado pela convivência diária. Apesar de não ser do jeito que gostaria (ainda), a minha vida é muito mais feliz pela presença de vocês!

À Helena Silberman que é a pessoa com o melhor coração que já conheci. Todo o seu carinho, preocupação, convivência e conversas foram/são de extrema importância para tornar a minha vida muito mais agradável. Nossas longas conversas por vídeo sempre alegravam o meu dia. Obrigado!

Ao meu orientador professor Marcos Vicente de Brito Moreira pelo trabalho realizado. Sempre acreditou em mim e me deu liberdade para produzir o que eu

imaginava. As discussões foram muito importantes para os aprimoramentos e qualidade final do trabalho. Além disso, mesmo quando eu comecei a trabalhar, não teve problemas em me orientar.

À Compagnie Générale de Géophysique (CGG) por ter me liberado temporariamente do trabalho para que eu pudesse focar na escrita. A empresa investe muito na educação do seu pessoal e isso faz a total diferença.

A todos aqueles que me ajudaram de qualquer maneira ao longo dessa jornada. E, por fim, à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro - Código de Financiamento 001 - durante parte da realização do mestrado.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

MODELO DE AUTÔMATO BASEADO EM EVENTOS PARA IDENTIFICAÇÃO DE SISTEMAS DE EVENTOS DISCRETOS COM O OBJETIVO DE IDENTIFICAÇÃO DE FALHAS

Thiago Henrique de Marreiros Cordeiro Machado

Junho/2022

Orientador: Marcos Vicente de Brito Moreira

Programa: Engenharia Elétrica

Neste trabalho, um método para a construção de um modelo obtido via identificação de sistemas a eventos discretos do tipo caixa-preta, chamado de Modelo em Autômato Baseado em Eventos (EBAM), e um método para a detecção de falhas usando esse modelo são apresentados. A maior vantagem do EBAM é que, diferente dos demais apresentados na literatura, ele é baseado nas mudanças de eventos no lugar de mudança de status de entrada e saída do controlador. Isso permite que os caminhos observados para a identificação do modelo possuam estados iniciais diferentes. Tal vantagem é muito importante para a aplicação prática na indústria, uma vez que muitos processos possuem estados iniciais e finais diferentes (não necessariamente formam um ciclo) e cada processo pode ter um estado inicial próprio. Uma outra vantagem é que o EBAM, em geral, é mais compacto do que o seu equivalente por estados. O modelo representa o comportamento livre de falhas do sistema (obtido por identidicação) e a detecção de falhas é feita comparando-se o comportamento atual do sistema com a estimativa do modelo. Caso haja alguma discrepância, uma falha é detectada através de cinco condições que determinam quando um evento é viável no modelo. Para cada método proposto é apresentado seu respectivo algoritmo e exemplos didáticos a fim de ilustrar o procedimento. Em seguida, para mostrar que é possível aplicar toda a teoria desenvolvida em sistemas reais, um exemplo prático é apresentado.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

EVENT-BASED AUTOMATON MODEL FOR IDENTIFICATION OF DISCRETE-EVENT SYSTEMS WITH THE AIM OF FAULT DETECTION

Thiago Henrique de Marreiros Cordeiro Machado

June/2022

Advisor: Marcos Vicente de Brito Moreira

Department: Electrical Engineering

In this work, a method for the construction of a model obtained by black-box discrete-event systems identification, called Event-Based Automaton Model (EBAM), and a method for fault detection using this model are presented. The main advantage of the EBAM is that, unlike the others present in the literature, it is based on event changes instead of input and output controller status changes. This allows the observed paths used to the identify the model to have different initial states. This advantage is an important detail for the practical application in industries, since many processes have different initial and final states (not necessarily closing a loop) and each process can have its own initial state. Another advantage is that the EBAM is, in general, more compact than its equivalent based on state changes. The model represents the fault-free behavior of the system (obtained by identification) and the fault detection is done by comparing the current behavior of the system with the estimated model. If there is any discrepancy, a fault is detected by using of five conditions that determine when an event is viable in the model. For each proposed method, its respective algorithm and didactic examples are presented in order to illustrate the procedure. Then, to show that it is possible to apply all the theory developed in real systems, a practical example is presented.

Contents

List of Figures

List of Tables

List of Symbols

k	Free parameter for the identification, p. 2
$\mathbb{Z}^+$	Set of positive integers, p. 2
$\|\|S\|\|$	Cardinality of the set S, p. 6
$\|w\|_s$	Number of symbols of the word w, p. 6
ε	Empty word, p. 6
Σ^*	Kleene-Closure of the set Σ, p. 7
$\overline{L}$	Prefix-closure of language L, p. 7
$:=$	The notation for "equal by definition", p. 7
!	“is defined”, p. 8
$\mathbb{Z}_2$	Set of possible values for a binary number, *i.e.*, $\{0, 1\}$, p. 10
$u[i]$	i-th element of vector u, p. 10
$u(i)$	i-th observation of vector u, p. 10
$\uparrow x$	Rising edge of the signal $u[x]$, where u is an I/O vector, p. 10
$\downarrow x$	Falling edge of the signal $u[x]$, where u is an I/O vector, p. 10
$\|p\|_p$	Length of the path p, p. 11
2^S	Power set of the set S, *i.e.*, the set formed of all subsets of S, p. 12
p_i^k	i-th modified path p, p. 13
$\|u\|_v$	Length of the vector u, p. 21
$\tilde{\sigma}$	Codificated event, where σ is an event, p. 23
$\tilde{s_i}$	EBAM event sequence associated to the path p_i, p. 23

$\tilde{s_i}^k$ EBAM modified sequence of events associated to the EBAM event sequence $\tilde{s_i}$, p. 24

$\tilde{p}$ EBAM path associated to the path p, p. 28

List of Abbreviations

DES	Discrete Event Systems, p. 1
NDAAO	Non-Deterministic Autonomous Automaton with Outputs, p. 2
PLC	Programmable Logic Controller, p. 2
I/O	Input/Output, p. 2
DAOCT	Deterministic Automaton with Outputs and Conditional Transitions, p. 3
EBAM	Event-Based Automaton Model, p. 3
3D	Three-Dimensional, p. 4
GUI	Graphical User Interface, p. 62

Chapter 1

Introduction

Fault diagnosis is an important problem that has been developed still nowadays by the scientific community. The first approach in Discrete Event Systems (DES) was presented in [1], where the concept of system diagnosability was defined. Roughly speaking, a system is diagnosable if it is possible to detect and isolate, within a bounded delay of events, occurrences of faults using the recorded observed events [1]. Using this concept, several works developed new theories and improvements [2–11]. However, the great majority of the methods proposed in the literature need the model and the post-fault behavior of the system in order to correctly diagnose the fault occurrence.

In general, real systems are large and composed of several subsystems, thus, for small ones, obtain their models and behaviors after faults by hand is possible, but, for complex systems it would be almost impossible, since there are a lot of behaviors to model and the post-fault behavior would lead the system to unpredictable states. In addition, only known faults would be detected and it is necessary an engineer who is familiar with DES and knows deeply the plant behavior to handle it. The modelling process is laborious, time consuming and the obtained model must be accurate. For these reasons, using the traditional approach for fault diagnosis in real systems can be a difficult task.

To overcome these problems, identification techniques have been proposed in the literature [13, 14]. It solves the problem of obtaining the model, but it brings the problem of how to obtain it in an accurately way. The core ideia is to reproduce the fault-free system behavior from the observation of the sequences of events generated by the system. In the DES case, the information from sensors and actuators is binary and the model can be generated using Petri nets or finite automata, for example. It is important to remark that the concept of diagnosability cannot be used, since the observed data corresponds only to the fault-free behavior of the system.

Several works proposed in the literature address the problem of DES identification using Petri nets [13, 15–22], however, most of them do not address the problem of fault diagnosis [13, 15, 16, 18, 20, 21] and when it is addressed [17, 19, 22], it is always necessary some knowledge about the system. In [17] it is assumed that partial knowledge about the model is known, for example, the cardinality of the subset of measurable places and an upper bound of the cadinality of the set of places. In [19] it is assumed that the fault-free system structure and dynamics are known. In [22] it is assumed that the fault-free behavior of the system is previously known since only the post-fault behavior is identified. In general, Petri nets are used when some information about the system is known, because they have more complex structure than automata and it allows to use this information in the model. For example, if it is known that there are some parts of the system that works in parallel, it can be added to the model using a concurrence structure. When there is no information about the system behavior or structure it is simpler to perform a black-box identification using automata due to their more basic structure.

Several methods are proposed in the literature for DES identification with the aim of fault diagnosis using automata [14, 23–25]. The identified models represent the fault-free behavior of the system and the main advantages of these methods are the reduced computacional cost and the identified model can be obtained without any knowledge of the system (behavior, structure or model). In these works, fault isolation is carried out from the residual analysis, *i.e.*, comparing the expected behavior (fault-free) with the observed one and analysing the discrepancy between them.

In [23, 24] the Non-Deterministic Autonomous Automaton with Outputs (NDAAO) is presented. The NDAAO was used for the identification of a closed-loop industrial DES using data read directly from the Programmable Logic Controller (PLC) and this scheme can be seen in Figure 1.1. The vector composed of the signals of the system, *i.e.*, sensors and actuators, is called Input/Output (I/O) vector. The I/O vector replacing the signals for its values, 0 or 1 since they are binary, at an observation is called I/O status. Paths are production cycles and they are represented by sequences of consecutives I/O status. The NDAAO is obtained using paths to generate the fault-free model. To balance size and accuracy of the model, it is used a free parameter $k \in \mathbb{Z}^+$, where $\mathbb{Z}^+$ denotes the set of positive integers, that defines the amount of past I/O status that is recorded in each state of the model. In general, small values of k are used with the objective of obtaining compact models and high ones are used with the objective of obtaining accurate models. The simplest model is when k is equal to 1, that assigns only one I/O status to each state of the model. The accuracy of a model is related to the exceeding language.

This language is formed of the sequences that can be reproduced in the model but cannot occur in the system and a large exceeding language reduces the capability of detecting faults since there may exist fault sequences that are represented in the model. The NDAAO satisfies the property of k - completeness (in the sense of [26]), *i.e.*, all and only all observed subwords of length smaller than or equal to k are represented in the model. In words, this means that all subsequences of length k were observed, thus, up to k observed events there are no exceeding language in the model. Therefore, increasing k increases the accuracy of the model.

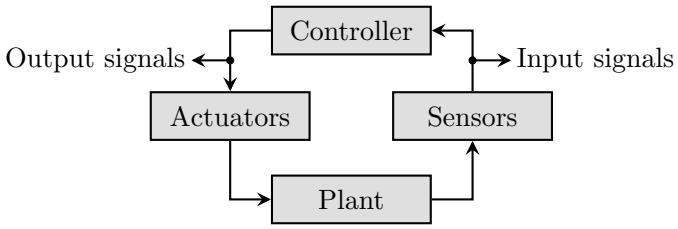

Figure 1.1: Signals exchanged between plant (system) and controller (PLC) in a closed-loop system.

In [23, 24], it is assumed that all paths start with the same I/O status. However, in several cases, the system may execute different processes starting with distinct initial I/O status. For these cases, it is not possible to apply the methods in [23, 24] to obtain the NDAAO of the system.

In [25] the Deterministic Automaton with Outputs and Conditional Transitions (DAOCT) is presented. It uses the same paths that are used to compute the NDAAO. In order to reduce the exceeding language, and, consequently, reduce the number of non-detectable faults, it is introduced the path estimation function. The number of possible next states in the model is reduced using the path estimation function, avoiding the model to go through states that are not possible to be executed by the system. If DAOCT is acyclic, then, the exceeding language is empty. In addition, it also holds the k - completeness property. It is important to remark that, as in the computation of the NDAAO, the DAOCT is computed from the paths assuming that all of them start with the same I/O status.

In this work, we present the Event-Based Automaton Model (EBAM) for DES identification with the aim of fault detection [12] which solves the main problem of the NDAAO and the DAOCT models: the dependence of the same initial I/O status for all paths. Instead of associating I/O status to the states, the EBAM uses vectors that represent the events (that are the difference between two consecutives I/O status). The EBAM, as the DAOCT, has some important properties: (i) it also uses a free parameter k, leading to a trade-off between model size and accuracy;

(*ii*) it also uses a path estimation function, reducing the exceeding language of the system model; (*iii*) the observed language is a subset of the identified language, *i.e.*, the EBAM simulates the observed fault-free system behavior; (*iv*) the model is k-complete, thus, any subword of length k belongs to the EBAM if, and only if, it has been observed and; (*v*) the exceeding language is the empty set if the model is acyclic. In addition, for a given value of k, the EBAM is, in general, more compact than the other models, which may increase the exceeding language, but we show that this increase is not significant in some cases. To illustrate the use of the EBAM we present a practical example using the three-dimensional (3D) simulation software Factory I/O and a Siemens PLC. In this example, the use of the NDAAO or the DAOCT is not possible, since the observed paths start with different initial I/O status.

This work is divided in 5 chapters, described in the sequel.

In Chapter 2 – Theoretical Background –, preliminary concepts about DES such as its definition, languages and automata are presented. At the end, the DAOCT is also presented to show its properties, limitations and to allow comparisons later.

In Chapter 3 – Event-Based Automaton Model –, an example is shown to motivate the EBAM creation. Later, the EBAM is formally defined as the procedure to fully construct it, its languages and properties. Finally, at the end, the theory and the procedure for fault detection using EBAM are also presented. In addition, some examples are shown to illustrate the use of the algorithms.

In Chapter 4 – Practical Example –, a practical example using the Factory I/O, to simulate the system, and the Siemens PLC S7-1200, as the controller, is presented to illustrate the application of the EBAM in a real situation. The model is computed and 44 fault scenarios are designed to test the fault detection using the EBAM. Then, some discussion about the results are presented.

In Chapter 5 – Conclusions –, the conclusions of the work and possible directions for future research are presented.

Chapter 2

Theoretical Background

In this chapter, the definition of DES, automata and languages are presented in Sections 2.1, 2.2 and 2.3, respectively. In addition, at the end of this chapter, in Section 2.4, the DAOCT model is presented.

2.1 Discrete-Event Systems

Systems can be classified regarding their dynamics as: (*i*) time-driven or (*ii*) event-driven. The first one is ruled by time and it can be modeled by differential equations, for continuous time systems, or difference equations, for discrete time systems. Roughly speaking, the passage of time drives the system. The system formed of a pump and a tank where the tank level is the system state is an example of a time-driven system. Given the initial condition (state), the system model and the input of the system, the tank level over the time can be computed for any given time instant after the initial time.

The dynamics of event-driven systems are governed by the occurrence (in general, asynchronous in time) of events and it cannot be modeled by differential or difference equations. There are some formalisms to model event-driven systems, where the most used are automata and Petri nets. When an event occurs, the model can evolve (instantaneously to a new state or to itself) or not. An elevator can be abstracted as an event-driven system. It may change its state (floor) when someone pushes any button. If the elevator is already in the desired floor, nothing is done, otherwise, the elevator goes to the desired floor. Since in DES all the events are discrete, continuous signals are discretized in order to model them. For example, in the elevator case, the pressing of the button can be considered as an event.

Now that the basic ideas of DES have been presented, its formal definition [27, 28] is presented in Definition 2.1.

Definition 2.1 (Discrete-Event Systems)
Discrete-event systems are dynamic systems such that: (i) the state space is a discrete set (not necessarily finite) and (ii) the evolution of the system is event-driven.

One possibility to study the DES logical behavior is using language theory and automata. These concepts are presented in Sections 2.2 and 2.3, respectively.

2.2 Languages

Like in any language, such as English or Portuguese, the concatenation of symbols of an alphabet generates words (sequences) and a set of words generates a language. In the DES case, as an abstraction, symbols are events, alphabets are set of events, words are sequences of events and subwords are subsequences of events. The formal definition of an alphabet is presented in Definition 2.2.

Definition 2.2 (Alphabet)
An alphabet Σ is a finite nonempty set of symbols.

For example, $\Sigma = \{a, b\}$ is an alphabet with two symbols, a and b, and its cardinality is $\|\Sigma\| = 2$.

The definition of a word is presented in the sequel. In the DES case, a word is a sequence of events.

Definition 2.3 (Word)
A word w is a sequence of symbols and its length $|w|_s$ is the number of events that form w, counting multiple occurrences of the same event. The empty word, represented by ε, is a special case whose length is $|\varepsilon|_s = 0$.

For example, $u = ab$, $v = ba$ and $w = aba$ are words, but u and v are different since the order is important. In these cases, $|u|_s = |v|_s = 2$ and $|w| = 3$.

The operation that constructs words is called concatenation and its definition is presented as follows.

Definition 2.4 (Concatenation)
The concatenation of the words u and v is an operation that generates a new word, respecting the order, consisting of the symbols of u immediately followed by the symbols of v, thus, uv. The empty word ε is the identity element of the concatenation, thus, for any word u, $\varepsilon u = u\varepsilon = u$.

For example, the concatenation between $u = ab$ and $v = ba$ is $uv = abba$, while the concatenation between v and u is $vu = baab$.

The next concept is of language, whose definition can be seen in Definition 2.5.

Definition 2.5 (Language)
A language L is a set of words of finite length.

For example, $L = \{a, b, aab, bbb\}$ and $K = \{a\}$ are languages, whose cardinalities are $\|L\| = 4$ and $\|K\| = 1$, respectively.

The operation over Σ that creates a language formed of all possible finite words obtained from Σ is called the *Kleen-Closure*, and its definition is presented in Definition 2.6.

Definition 2.6 (Kleene-Closure)
Let Σ be a set of events, then, $\Sigma^* := \{\varepsilon\} \cup \Sigma \cup \Sigma\Sigma \cup \Sigma\Sigma\Sigma \dots$ is the Kleene-Closure of Σ and it is defined as the set of all finite words formed with the elements of Σ, including ε.

The notation $:=$ means "equal by definition". And, a property of the Kleene-Closure is that Σ^* is infinite, but countable.

For example, for $\Sigma = \{a, b\}$, we have that $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab \dots\}$.

It is important to remark that every language formed of elements of Σ is a subset of Σ^*, since Σ^* has all possible words generated from Σ.

Every word w can be decomposed into three parts: (i) prefix, (ii) subword and (iii) suffix. Let $\Sigma = \{a, b, c\}$ be an alphabet and $w = abc$ be a word. Therefore:

- ε, a, ab and abc are all the prefixes of w;
- ε, a, b, c, ab, bc and abc are all the subwords of w;
- ε, c, bc and abc are all the suffixes of w.

Note that ε and w are always a prefix, a subword and a suffix of w.

Since languages are sets, the operations performed on sets are valid in language theory, such as union, difference and intersection. Another operation that can be defined for language is called Prefix-Closure and it creates a new language consisting of all the prefixes of all the words of the original language.

Definition 2.7 (Prefix-Closure)
Let $L \subseteq \Sigma^*$ be a language, then, $\overline{L} := \{s \in \Sigma^* : (\exists t \in \Sigma^*)[st \in L]\}$ is the prefix-closure of L.

Notice that $L \subseteq \overline{L}$. In addition, if a language L is equal to its prefix-closure, *i.e.*, $L = \overline{L}$, then, L is said to be prefix-closed.

For example, let $\Sigma = \{a, b, c\}$ be an alphabet and $L_1 = \{\varepsilon, a, ab\}$ and $L_2 = \{a, b, ac, aab\}$ be two languages over Σ. Then, $\overline{L_1} = \{\varepsilon, a, ab\}$ and $\overline{L_2} = \{\varepsilon, a, b, aa, ac, aab\}$. Note that since $L_1 = \overline{L_1}$, then, L_1 is prefix-closed.

2.3 Automata

In computation theory, automaton is a formalism used to represent languages and it is represented graphically by the state transition diagram. Automaton is one of the approaches most used to model DES because it can reproduce the DES behavior using a compact representation. The formal definition of a Deterministic Finite State Automaton, which will be refered to only as Automaton when the context is clear, is presented in Definition 2.8 [27].

Definition 2.8 (Deterministic Finite State Automaton)
A Deterministic Finite State Automaton G is a 5-tuple

$$G := (X, \Sigma, f, x_0, X_m)$$

where:

- X *is the finite set of states;*
- Σ *is the finite set of events;*
- $f : X \times \Sigma \to X$ *is the transition function (it can be partially defined over its domain);*
- $x_0 \in X$ *is the initial state;*
- $X_m \subseteq X$ *is the set of marked states.*

Let $x, y \in X$ and $\sigma \in \Sigma$, then $f(x, \sigma) = y$ means that at state x, after the occurence of event σ, the automaton will change its state to y. In this case, the function is defined at x, σ and this can be represented as $f(x, \sigma)!$, where ! means "is defined". This function can be extended in a natural way as $f : X \times \Sigma^* \to X$ where

$$\begin{cases} f(x, \varepsilon) := x \\ f(x, s\sigma) := f(f(x, s), \sigma),\ s \in \Sigma^* \wedge \sigma \in \Sigma. \end{cases}$$

The set of marked states is, in general, a set of states whose information is relevant, such as the accomplishment of a task.

Automata are graphically represented by state transition diagrams which are directed graphs, where the vertices are the states and the edges are the transitions, labeled with the events that causes the changing in the states. In addition, the initial state is marked by an arrow (without an origin state). Example 2.1 presents an automaton and its graphical representation.

Example 2.1 (Example of an Automaton)
Let $G = (X, \Sigma, f, x_0, X_m)$ *be an automaton such that:*

- $X = \{x_0, x_1, x_2, x_3\}$;
- $\Sigma = \{a, b, c, d, e\}$;
- $f(x_0, a) = x_1$, $f(x_1, b) = x_3$, $f(x_1, d) = x_2$, $f(x_2, b) = x_3$, $f(x_2, e) = x_0$, $f(x_3, c) = x_1$, $f(x_3, b) = x_3$;
- $x_0 = x_0$;
- $X_m = \{x_3\}$.

The state transition diagram of G *is depicted in Figure 2.1.*

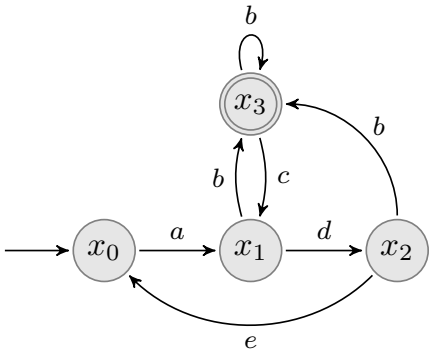

Figure 2.1: Automaton of the Example 2.1.

The language generated by an automaton G is the set composed of all the words generated by G and it is defined as $\mathcal{L}(G) := \{s \in \Sigma^* : f(x_0, s)!\}$.

Similarly, the language marked by an automaton G is the set formed of the words that reach a marked state in G from its initial state and it is defined as $\mathcal{L}_m(G) := \{s \in \Sigma^* : f(x_0, s)! \wedge f(x_0, s) \in X_m\}$.

2.4 Deterministic Automaton with Outputs and Conditional Transitions

In order to define the Deterministic Automaton with Outputs and Conditional Transitions (DAOCT), it is first necessary to present some notations.

- $\mathbb{Z}^+$ is the set of positive integers numbers;
- $\mathbb{Z}_2 = \{0, 1\}$ is the set of possible values for a binary number.

The DAOCT [25] model is obtained from observations of the system behavior whose scheme is shown in Figure 1.1. The sensors are related to the inputs (I) of the PLC and the actuators are related to the outputs (O) of the PLC. Each sensor and actuator is represented by a subscribed number ($\in \mathbb{Z}^+$), for example, I_1 and O_3, correspond to the first input and the third output, respectively. The observed data are based on I/O vectors, defined as follows.

Definition 2.9 (I/O Vector)
The I/O vector $u \in \mathbb{Z}_2^n$ is a vector of signals where $n \in \mathbb{Z}^+$ is the number of sensors (inputs) and actuators (outputs) of the PLC.

The i-th element of a vector u is denoted by $u[i]$ where $i \in \mathbb{Z}^+$. For example, $u[2]$ is the second element of the I/O vector u. The value of the I/O vector u at a given observation $i \in \mathbb{Z}^+$ is called I/O status and is denoted as $u(i)$. An observation is acquired when there is a change in at least one signal of u.

Let $u = [I_1 \ \ I_2 \ \ O_1 \ \ O_2]^T$ be an I/O vector and $u(1) = [1\ 0\ 1\ 1]^T \in \mathbb{Z}_2^4$ its status (at the first observation). For example, $u[1]$ is the first element of the I/O vector u, that is I_1. And, $u(1)[2]$ is the second element of the I/O status $u(1)$, that is 0.

A notation used to simplify an I/O status which its elements are just 0 or 1 is to use tuples which the elements are the indexes (in ascending order) of the 1 values. For example, $u(1) = (1, 3, 4)$. In particular, if all elements of an I/O status are 0, it is represented as an empty tuple, *i.e.*, ().

Systems evolve via signal changes in the I/O status which are called events and their formal definition is presented in Definition 2.10 [25].

Definition 2.10 (Event)
An event of the identified model is any observed instantaneous change in one or more signals of the I/O status.

To simplify the representation of an event σ, we use $\uparrow$ to denote a rising edge and $\downarrow$ to denote a falling edge plus the index of the signal in the I/O vector. For

example, let $u = [I_1 \;\; I_2 \;\; O_1 \;\; O_2]^T$ be the I/O vector and $u(1) = [0\ 0\ 1\ 1]^T$ and $u(2) = [1\ 1\ 0\ 1]^T$ be the first two observed I/O status. Then, the first and the second element of u are rising edges while the third one is a falling edge, which can be represented as $\sigma(1) = \uparrow 1 \uparrow 2 \downarrow 3$.

Let $u(i)$ and $u(i+1)$ where $i \in \mathbb{Z}^+$ be two consecutive observed I/O status and $\sigma(i)$ be the event that caused $u(i)$ be changed to $u(i+1)$, then, this can be represented as $(u(i), \sigma(i), u(i+1))$. For example, $\left([0\ 0\ 1\ 1]^T,\ \uparrow 1 \uparrow 2 \downarrow 3,\ [1\ 1\ 0\ 1]^T\right)$. The extension of this representation leads us to the definition of paths, that can be seen in Definition 2.11 [25].

Definition 2.11 (Path)
A path is a sequence of I/O status and events. Let u be the I/O vector and $\sigma(i)$ be the event that causes the change between two consecutives I/O status $u(i)$ and $u(i+1)$, where $i \in \mathbb{Z}^+$. Then, for $l \in \mathbb{Z}^+$ observations, a path p is

$$p := (u(1), \sigma(1), u(2), \sigma(2), \ldots, \sigma(l-1), u(l)).$$

Its length $|p|_p$ is the number of observations, then, $|p|_p = l$.

To represent several paths in a compact way the subindex notation is introduced, where each path p_q represents the q-th path and its notation is $p_q := (u_q(1), \sigma_q(1), u_q(2), \sigma_q(2), \ldots, \sigma_q(l_q - 1), u_q(l_q))$, for $l, q \in \mathbb{Z}^+$. In addition, the initial I/O status of an I/O vector u is denoted as $u_{0_q} = u_q(1)$.

A path represents, for example, the prodution cycle in a system and there always exist a sequence of events associated to it. This sequence is formed of all the events that were recorded in the path. Let $p = (u(1), \sigma(1), u(2), \sigma(2), \ldots \sigma(l-1), u(l))$ where $l \in \mathbb{Z}^+$ be a path, then, the associated sequence of events is $s := \sigma(1)\sigma(2)\ldots\sigma(l-1)$. Thus, associated with each path p_q there is a sequence of events s_q.

In [25], the following assumptions are considered:

A1. Every path has the same initial I/O status, *i.e.*, the system has a unique initial state;

A2. A sequence of events associated to a path cannot be a prefix of another sequence of events associated with a different path.

Assumption A2. is expected to be true in several systems, since each path is a task in the system, and, in general, a task is not a subtask of another task.

Now the definition of the DAOCT is presented in Definition 2.12 [25].

Definition 2.12 (Determinitisc Automaton with Outputs and Conditional Transitions)
The Determinitisc Automaton with Outputs and Conditional Transitions M is a 9-tuple

$$M := (X, \Sigma, \Omega, f, \lambda, R, \theta, x_0, X_f)$$

where:

- X *is the set of states;*
- Σ *is the set of events;*
- $\Omega \subseteq \mathbb{Z}_2^n$ *is the set of all observed I/O status where* n *is the number of signals of the system;*
- $f : X \times \Sigma^* \to X$ *is the deterministic transition function;*
- $\lambda : X \to \Omega$ *is the state output function;*
- $R := \{1, 2, \ldots, r\}$ *is the set of path indexes where* $r \in \mathbb{Z}^+$*;*
- $\theta : X \times \Sigma \to 2^R$ *is the path estimation function;*
- x_0 *is the initial state;*
- $X_f \subseteq X$ *is the set of final states.*

Set Ω is composed of all observed I/O status. λ is the function that associates an I/O status to each state of the system. θ is the path estimation function which avoids the execution of sequences of events in wrong paths and this will be explained and illustrated in Example 2.3.

Before introducing the language generated by the DAOCT, it is necessary to extend the path estimation function θ to θ_s, to allow sequences of events instead of only events. Therefore, the domain is extended to $\theta_s : X \times \Sigma^* \to 2^R$. Then, the θ_s function is defined recursively as:

$$\theta_s(x, \varepsilon) = R,$$

$$\theta_s(x, s\sigma) = \begin{cases} \theta_s(x, s) \cap \theta(x', \sigma), \text{ where } x' = f(x, s), \text{ if } f(x, s\sigma)! \\ \text{Undefined} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{, otherwise.} \end{cases}$$

Finally, the language generated by the DAOCT is given by:

$$L(DAOCT) := \{s \in \Sigma^* : f(x_0, s)! \wedge \theta_s(x_0, s) \neq \emptyset\}.$$

The DAOCT is obtained from the observed paths of the system, and depends on a free parameter k. This parameter assigns at most k I/O status to each state of the model, that leads to a trade-off between accuracy and model size. High values for k imply in more accurate models at the cost of the model size increase, while small values of k imply the opposite. The construction of the modified paths is presented in Definition 2.13.

Definition 2.13 (Modified Paths)
Let p_q be the observed paths, u_q be I/O vectors, l_q be the length of p_q, σ_q be events where $q \in \mathbb{Z}^+$ and k be the free parameter. The modified path p_q^k is given as

$$p_q^k := (y_q(1),\, \sigma_q(1),\, y_q(2),\, \sigma_q(2),\, \ldots,\, \sigma_q(l_q - 1),\, y_q(l_q))$$

where, for $i \in \mathbb{Z}^+$:

$$y_q(i) := \begin{cases} (u_q(i-k+1),\, \ldots,\, u_q(l)), & \text{if } k \leq l \leq l_q \\ (u_q(1),\, \ldots,\, u_q(l)) & \text{, if } l < k. \end{cases}$$

Note that the free parameter k only changes how many I/O status will be associated to each vertex in the path. Thus, for $k = 1$, the path p^1 is equal to path p. To illustrate the construction of a modified path, an example will be given in the sequel.

Example 2.2 (Modified Paths Construction)

Suppose that the following paths have been observed:

$$p_1 = \left(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}, \uparrow 1, \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \uparrow 2, \begin{bmatrix}1\\1\\0\\0\end{bmatrix}, \uparrow 3, \begin{bmatrix}1\\1\\1\\0\end{bmatrix}, \downarrow 2 \downarrow 3, \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \downarrow 1, \begin{bmatrix}0\\0\\0\\0\end{bmatrix} \right)$$

$$p_2 = \left(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}, \uparrow 1, \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \uparrow 2, \begin{bmatrix}1\\1\\0\\0\end{bmatrix}, \uparrow 4, \begin{bmatrix}1\\1\\0\\1\end{bmatrix}, \downarrow 2 \downarrow 4, \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \downarrow 1, \begin{bmatrix}0\\0\\0\\0\end{bmatrix} \right)$$

$$p_3 = \left(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}, \uparrow 1, \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \downarrow 1, \begin{bmatrix}0\\0\\0\\0\end{bmatrix} \right)$$

Let us construct the modified paths considering $k = 2$. Then, the modified paths are:

$$p_1^2 = \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \uparrow 1, \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \uparrow 2, \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \uparrow 3, \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \downarrow 2 \downarrow 3, \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}, \downarrow 1, \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \right)$$

$$p_2^2 = \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \uparrow 1, \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \uparrow 2, \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \uparrow 4, \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, \downarrow 2 \downarrow 4, \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}, \downarrow 1, \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \right)$$

$$p_3^2 = \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \uparrow 1, \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \downarrow 1, \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \right)$$

Note that the initial I/O status is the same for a path p and a modified path p^k. In addition, the i-th I/O status of p^k has i columns if $i < k$, otherwise it has k columns.

In [25] an identification algorithm is proposed for the construction of the DAOCT model given the set of observed paths. To ilustrate its application, the model for the paths of Example 2.2 is presented in Example 2.3.

Example 2.3 (DAOCT Model)
Considering the paths of Example 2.2 and the I/O status $[1\ 1\ 1\ 0]^T$ *and* $[1\ 1\ 0\ 1]^T$ *as marked states, let us construct the models for the free parameter* $k = 1$ *and* $k = 2$.

For $k = 1$*, the constructed model* M_1 *is shown in Figure 2.2.*

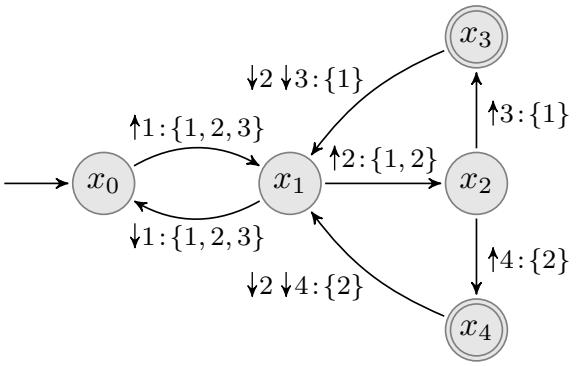

Figure 2.2: DAOCT model for $k = 1$ of the Example 2.2.

Let x_a and x_b be states, where $a, b \in \mathbb{Z}$, e be the event that evolves x_a to x_b and I be the set of all allowed path indexes associated with this evolution. Then, in state transitions, these informations are represented as $e : I$ on the arc that connects x_a to x_b.

For $k = 2$, the constructed model M_2 is shown in Figure 2.3.

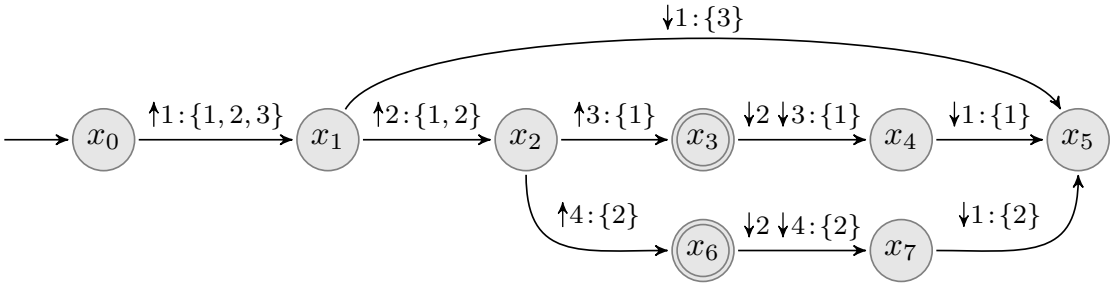

Figure 2.3: DAOCT model for $k = 2$ of the Example 2.2.

The DAOCT model of Figure 2.2, obtained for $k = 1$, has the following attributes:

- $X = \{x_0, x_1, x_2, x_3, x_4\}$;
- $\Sigma = \{\uparrow 1, \downarrow 1, \uparrow 2, \uparrow 3, \uparrow 4, \downarrow 2 \downarrow 3, \downarrow 2 \downarrow 4\}$;
- $\Omega = \{[0\ 0\ 0\ 0]^T, [1\ 0\ 0\ 0]^T, [1\ 1\ 0\ 0]^T, [1\ 1\ 1\ 0]^T, [1\ 1\ 0\ 1]^T\}$;
- $f(x_0, \uparrow 1) = x_1,\ f(x_1, \downarrow 1) = x_0,\ f(x_1, \uparrow 2) = x_2,\ f(x_2, \uparrow 3) = x_3,$ $f(x_2, \uparrow 4) = x_4,\ f(x_3, \downarrow 2 \downarrow 3) = x_1,\ f(x_4, \downarrow 2 \downarrow 4) = x_1$;
- $\lambda(x_0) = [0\ 0\ 0\ 0]^T,\ \lambda(x_1) = [1\ 0\ 0\ 0]^T,\ \lambda(x_2) = [1\ 1\ 0\ 0]^T,\ \lambda(x_3) = [1\ 1\ 1\ 0]^T,$ $\lambda(x_4) = [1\ 1\ 0\ 1]^T$;
- $R = \{1, 2, 3\}$;
- $\theta(x_0, \uparrow 1) = \{1, 2, 3\},\ \theta(x_1, \downarrow 1) = \{1, 2, 3\},\ \theta(x_1, \uparrow 2) = \{1, 2\},$ $\theta(x_2, \uparrow 3) = \{1\},\ \theta(x_2, \uparrow 4) = \{2\},\ \theta(x_3, \downarrow 2 \downarrow 3) = \{1\},\ \theta(x_4, \downarrow 2 \downarrow 4) = \{2\}$;
- $x_0 = x_0$ *(It is important to highlight that x_0 is commonly used as the attribute of the model and as the name of the initial state);*
- $X_f = \{x_3, x_4\}$.

The DAOCT model of the Figure 2.2, $k = 2$, has the following attributes:

- $X = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$;
- $\Sigma = \{\uparrow 1, \downarrow 1, \uparrow 2, \uparrow 3, \uparrow 4, \downarrow 2 \downarrow 3, \downarrow 2 \downarrow 4\}$;
- $\Omega = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} \right\}$;
- $f(x_0, \uparrow 1) = x_1,\ f(x_1, \downarrow 2) = x_2,\ f(x_1, \downarrow 1) = x_5,\ f(x_2, \uparrow 3) = x_3,$
  $f(x_2, \uparrow 4) = x_6,\ f(x_3, \downarrow 2 \downarrow 3) = x_4,\ f(x_4, \downarrow 1) = x_5,\ f(x_6, \downarrow 2 \downarrow 4) = x_7,$
  $f(x_7, \downarrow 1) = x_5$;
- $\lambda(x_0) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},\ \lambda(x_1) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix},\ \lambda(x_2) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix},\ \lambda(x_3) = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix},$

  $\lambda(x_4) = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix},\ \lambda(x_5) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix},\ \lambda(x_6) = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix},\ \lambda(x_7) = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}$;
- $R = \{1, 2, 3\}$;
- $\theta(x_0, \uparrow 1) = \{1, 2, 3\},\ \theta(x_1, \downarrow 2) = \{1, 2\},\ \theta(x_1, \downarrow 1) = \{3\},\ \theta(x_2, \uparrow 3) = \{1\},$
  $\theta(x_2, \uparrow 4) = \{2\},\ \theta(x_3, \downarrow 2 \downarrow 3) = \{1\},\ \theta(x_4, \downarrow 1) = \{1\},\ \theta(x_6, \downarrow 2 \downarrow 4) = \{2\},$
  $\theta(x_7, \downarrow 1) = \{2\}$;
- $x_0 = x_0$ *(It is important to highlight that x_0 is commonly used as the attribute of the model and as the name of the initial state);*
- $X_f = \{x_3, x_6\}$.

Notice that despite of the difference in the value of k, M_1 and M_2 share the same set of events Σ, the set of path indexes R and the initial state x_0.

Now, let $s_1 = (\uparrow 1, \uparrow 2, \uparrow 3, \downarrow 2 \downarrow 3, \downarrow 1)$, $s_2 = (\uparrow 1, \uparrow 2, \uparrow 3, \downarrow 2 \downarrow 3, \uparrow 2)$ and $s_3 = (\uparrow 1, \uparrow 2, \uparrow 3, \downarrow 2 \downarrow 3, \uparrow 2, \uparrow 4)$ be sequences of events. The models M_1 and M_2 accept s_1, however, only M_1 accepts s_2 and none of them accept s_3. Even s_3 being reproducible in M_1, the set of possible path indexes is empty, therefore, there is no path that reproduces s_3. This means s_3 is part of the original exceeding language, but with the path estimation function it does not belong to the model language and it will be identified as a faulty sequence. The main goal of the path estimation function

is to reduce the exceeding language not allowing sequences that have empty sets of possible path indexes. To exemplify how this works, the evolution of the set of the possible path indexes θ_s is shown below. In the beginning, the model is at state x_0 with $\theta_s = R = \{1, 2, 3\}$. As the system evolves, θ_s is updated and this is shown below:

$x_0, \uparrow 1 \Rightarrow \theta_s = \theta_s \cap \theta(x_0, \uparrow 1) = \{1, 2, 3\} \cap \{1, 2, 3\} = \{1, 2, 3\}$
$x_1, \uparrow 2 \Rightarrow \theta_s = \theta_s \cap \theta(x_1, \uparrow 2) = \{1, 2, 3\} \cap \{1, 2\} = \{1, 2\}$
$x_2, \uparrow 3 \Rightarrow \theta_s = \theta_s \cap \theta(x_2, \uparrow 3) = \{1, 2\} \cap \{1\} = \{1\}$
$x_3, \downarrow 2 \downarrow 3 \Rightarrow \theta_s = \theta_c \cap \theta(x_3, \downarrow 2 \downarrow 3) = \{1\} \cap \{1\} = \{1\}$
$x_1, \uparrow 2 \Rightarrow \theta_s = \theta_s \cap \theta(x_1, \uparrow 2) = \{1\} \cap \{1, 2\} = \{1\}$
$x_2, \uparrow 4 \Rightarrow \theta_s = \theta_s \cap \theta(x_2, \uparrow 4) = \{1\} \cap \{2\} = \emptyset$

Since the last $\theta_s = \emptyset$, the evolution is not allowed, then, this sequence of events is not accepted.

In fact, s_2 and s_3 were not observed in the paths, therefore, they are expected to be faulty sequences. The model M_2 ($k = 2$) is more accurate than M_1 ($k = 1$), since it does not accept s_2 and s_3 while M_1 accepts s_2. This has happened because M_1 has cycles and this has introduced possible sequences of events that were not observed. However, the accuracy of M_2 is achieved increasing the size of the model (8 states against 5 states of M_1).

The DAOCT model has the following important properties as presented in [25]:

P1. The language of the DAOCT is a subset of the observed language;

P2. The DAOCT model is k-complete;

P3. If the DAOCT model does not have cycles for a given value of the free parameter k, then, the exceeding language is the empty set.

Property **P1** states that the DAOCT model simulates the observed fault-free behavior. In words, all the observed language belongs to the language generated by the model. Property **P2** states that for a given value of k, any subword of length k belongs to the DAOCT model if, and only if, it has been observed, which implies that the DAOCT model is suitable for fault detection.

In order to achieve the result of Property **P3**, the value of k is increased to reduce the cycles. This strategy is shown in Example 2.3, where the DAOCT model for $k = 1$ has cycles, differently from the model for $k = 2$. The main problem of this trade-off between model size and accuracy is how to obtain the most compact model that is capable of identifying faults. However, in several practical cases, it has been observed that small values for k can be chosen [24].

Chapter 3

Event-Based Automaton Model

In this chapter, an example to motivate the use of the EBAM is presented in Section 3.1. Then, in Section 3.2, the EBAM is formally introduced, as well as the procedure to compute the model and the algorithm to detect and isolate faults. Finally, the EBAM properties are presented in Section 3.4.

3.1 Motivating Example

Consider the box filling system depicted in Figure 3.1. This system is composed of a conveyor, a balance, containers to be filled (boxes), a tank and water (to fill the boxes). Its objective is to fill the boxes with a certain amount of water. There are four sensors i_1, i_2, i_3 and i_4 and three actuators o_1, o_2 and o_3 that control the system and are described in Table 3.1.

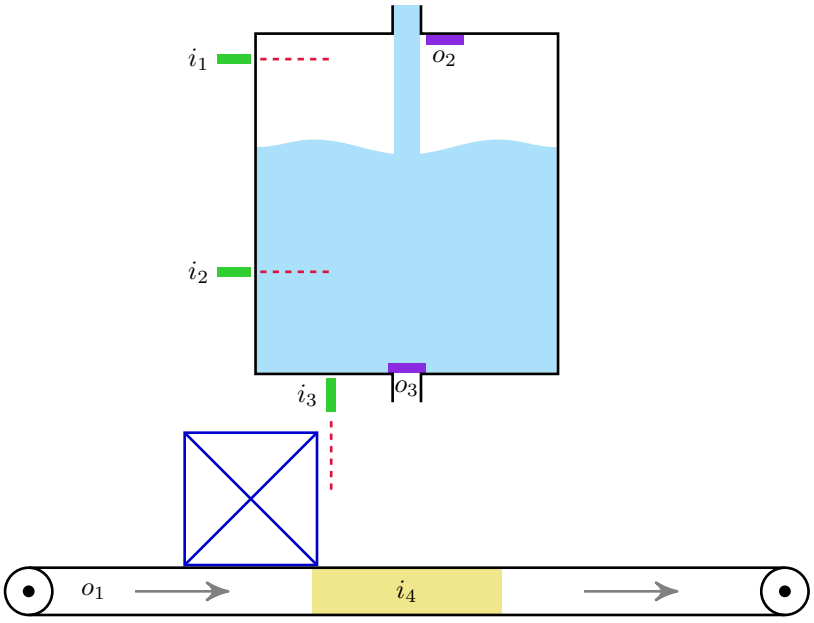

Figure 3.1: A box filling system.

Table 3.1: Description of each signal of the system in Figure 3.1.

Signal	Type	Description
i_1	P Sensor	Detect that the water level reached the maximum level
i_2	P Sensor	Detect that the water level reached the minimum level
i_3	P Sensor	Detect that there is a box entering in the balance
i_4	P Sensor	Detect when a box is filled (by the weight)
o_1	Actuator	Turn on and off the conveyor
o_2	Actuator	Open (1) and close (0) the valve that supplies the tank
o_3	Actuator	Open (1) and close (0) the valve that fills the boxes

Regarding the signal i_4, it is important to note that this signal is discretized, since the balance measures the weight. If the weight measured is greater than or equal to the amount necessary to fill the boxes, its values is 1, otherwise, its value is 0.

The general behavior of the system is described in steps as follows:

S1. The conveyor is turned on;

S2. A box arrives in the balance;

S3. The conveyor is turned off;

S4. The tank starts to fill the box;

S5. The box is completely filled;

S6. Back to step **S1**.

Besides the general behavior, the tank has its own behavior and it is described as follows:

- Only at the beginning, if the water level is lower than the maximum one, then, the tank is supplied with water, otherwise it is not.
- If the water level reachs the maximum one, then, the tank supply is interrupted.
- If the water level reachs the minimum level, then, the tank is supplied with water untill the water level reachs the maximum one.

Consider now $u = [i_1 \ i_2 \ i_3 \ i_4 \ o_1 \ o_2 \ o_3]^T$ as the I/O vector that will be used to observe the system.

It is important to remark that this sytem may have different initial I/O status, since the tank has its own behavior, then, at the beginning, the tank may be being supplied or not. This behavior also may cause different initial and final I/O status in the paths. Therefore, it is not possible to obtain a unique NDAOO or DAOCT model.

Let p_1, p_2 and p_3 be three observed paths of the system (splitted after a single observation). The observation started with the I/O status $u(1) = [0\ 1\ 0\ 0\ 0\ 0\ 0]^T$, *i.e.*, the water level was between the minimum and maximum level, in this case, no box is at the conveyor and all actuators are turned off (conveyor and valves).

$$p_1 = \left(\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \uparrow 5 \uparrow 6, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \uparrow 3, \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \uparrow 7 \downarrow 3 \downarrow 5, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \uparrow 4 \uparrow 5 \downarrow 7, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \right)$$

$$p_2 = \left(\begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \downarrow 4, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \uparrow 1 \downarrow 6, \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \uparrow 3, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \uparrow 7 \downarrow 3 \downarrow 5, \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \downarrow 1, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \uparrow 4 \uparrow 5 \downarrow 7, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \right)$$

$$p_3 = \left(\begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \downarrow 4, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \uparrow 3, \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \uparrow 7 \downarrow 3 \downarrow 5, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \uparrow 4 \uparrow 5 \downarrow 7, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \right)$$

Path p_1 represents the first task of the system. It starts with the water level lower than the maximum one, thus, the tank starts to be supplied. In addition, the conveyor is turned on. After a while, the sensor i_3 detects the presence of a box,

marked by its falling edge, then, the conveyor is turned off and the box starts to be filled. Finally, when the rising edge of the sensor i_4 occurs, that is, the box is already filled, then, the filling is interrupted and the conveyor is turned on.

Path p_2 represents the second task of the system, which comes right after the first one. The water level is still lower than the maximum one, thus the supply continues. After a while, the balance is empty, since the conveyor is on, and it is represented by the falling edge of sensor i_4. Then, the water level reachs the maximum level, detected by the rising edge of sensor i_1, then, the tank supply is interrupted. At a subsequent time, sensor i_3 detects the presence of a new box, marked by its falling edge, then, the conveyor is turned off and the box starts to be fill. In filling the box, the water level descreases and it is lower than the maximum one and it is detected by the falling edge of sensor i_1. Finally, the same process of releasing the filled box from the path p_1 is executed.

Path p_3 represents the third task of the system, which comes right after the second one. The difference between p_2 and p_3 is that in p_3 the process of stop the tank filling is not observed, since it has already stopped.

Note that all the paths have different initial I/O status. Furthermore, the initial and final I/O status of p_1 and p_2 are different. Therefore, as mentioned earlier, the NDAAO and the DAOCT are not suitable to model this system.

Notice that the same event can be associated to different I/O status transititons. For example, the event $\downarrow 4$ is associated to the transition from $[0\ 1\ 0\ 1\ 1\ 1\ 0]^T$ to $[0\ 1\ 0\ 0\ 1\ 1\ 0]^T$ (p_2) and from $[0\ 1\ 0\ 1\ 1\ 0\ 0]^T$ to $[0\ 1\ 0\ 0\ 1\ 0\ 0]^T$ (p_3).

3.2 Presentation of the Model

Before introducing the EBAM definition, some concepts and notations will be given. The concept of event, Definition 2.10, is the same used in the DAOCT. However, differently from the DAOCT that uses I/O status to create the model, the EBAM uses event vectors, which definition can be seen in the sequel.

Definition 3.1 (Event Vector)
Let u be an I/O vector and $u(i)$ and $u(i+1)$ be two consecutives I/O status where $i \in \mathbb{Z}^+$. Then, an event vector $e(i)$ is given by

$$e(i) := u(i+1) - u(i).$$

Its length $|e(i)|_v$ is the length of u, thus, $|e(i)|_v = |u|_v$. In addition, $e(i)[j]$ is its j-th element, where $j \leq |e(i)|_v$, $j \in \mathbb{Z}^+$.

Note that each element of an event vector $e(i)$ can assume only three values, since the I/O vector is formed by discrete signals. These values are: (i) -1, when $e(i)[j] = 1$ and $e(i)[j+1] = 0$; (ii) 0, when $e(i)[j] = e(i)[j+1] = 0$ or 1; (iii) 1, when $e(i)[j] = 0$ and $e(i)[j+1] = 1$, where $i, j \in \mathbb{Z}^+$. Now, let $\mathbb{1}$ be the set formed by these values, thus, $\mathbb{1} := \{\text{-1, 0, 1}\}$. Hence, an event vector $e(i) \in \mathbb{1}^{|e(i)|_v}$.

The model proposed in this work for the identification of DES aiming fault detection is called EBAM, which definition can be seen in Definition 3.2.

Definition 3.2 (Event-Based Automaton Model)
An Event-Based Automaton Model M is an 8-tuple:

$$M := (X,\, x_0,\, \Sigma,\, \delta,\, \lambda,\, \lambda_0,\, \Omega,\, \theta)$$

where:

- X *is the finite set of states;*
- $x_0 \in X$ *is the initial state;*
- Σ *is the finite set of events;*
- $\delta : X \times \Sigma \to X$ *is the transition function;*
- $\lambda_0 : \{x_0\} \to 2^{\mathbb{Z}_2^n}$ *is the initial state label function;*
- $\lambda : X \setminus \{x_0\} \to \bigcup_{i=1}^{k} 2^{\mathbb{Z}_2^{n^i}}$ *is the state label function;*
- Ω *is the finite set of path indexes;*
- $\theta : X \times \Sigma \to 2^{\Omega}$ *is the path indexes function;*

Function λ_0 associates the initial state to a set formed of the initial I/O status of the paths. The λ function associates to each state, except for the initial one, a modified event sequence, which is later defined in Definition 3.4. Ω is the set formed by the indexes associated to each path used to construct the model. The θ function is the same used in the DAOCT. In addition, the θ_s function is analogously defined.

First, it is necessary to extend the path estimation function, θ to θ_s, to allow sequences of events instead of only events. Therefore, the domain is extended to $\theta_s : X \times \Sigma^* \to 2^{\Omega}$. Then, the θ_s function is defined recursively as:

$$\theta_s(x,\, \varepsilon) = \Omega,$$

$$\theta_s(x,\, s\sigma) = \begin{cases} \theta_s(x,\, s) \cap \theta(x',\, \sigma), \text{ where } x' = \delta(x,\, s), \text{ if } \delta(x,\, s\sigma)! \\ \text{Undefined} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{, otherwise.} \end{cases}$$

It is important to highlight that the model is deterministic, since δ is deterministic and there is only one initial state. In addition, the model is based on event changes (noticed by the event vector), not in I/O status. This can be noted by the λ function. In DAOCT, it returns the I/O status (modified) associated to the state, however, in EBAM, it returns the event vector (modified) associated to the state.

An event is a change in one or more signals of an I/O vector and this can be represented as a vector using the bijective codification function f_c, which definition can be seen in Definition 3.3.

Definition 3.3 (Codification Function)
Given an I/O vector u, the bijective codification function f_c is:

$$f_c : \Sigma \to \mathbb{1}^{|u|_v}$$
$$\sigma \mapsto \tilde{\sigma}$$

where:

$$\tilde{\sigma}[i] := \begin{cases} 1, \text{ if } \uparrow u[i] \in \sigma \\ -1, \text{ if } \downarrow u[i] \in \sigma, \, i \in \{1, 2, \ldots, |u|_v\} \\ 0, \text{ otherwise.} \end{cases}$$

For example, let $u = [i_1 \;\; i_2 \;\; o_1 \;\; o_2]^T$ be an I/O vector, where i_1, i_2, o_1 and o_2 are signals and $\sigma = \uparrow 1 \uparrow 3 \downarrow 2$ be an event. Then, the codified event $\tilde{\sigma}$ is obtained by $f_c(\sigma)$. Hence, $\tilde{\sigma} = f_c(\sigma) = [1 \;\; -1 \;\; 1 \;\; 0]^T$. Now, let $\tilde{\sigma} = [0 \;\; 0 \;\; 1 \;\; -1]^T$, then, σ is obtained by $f_c^{-1}(\tilde{\sigma})$. Hence, $\sigma = f_c^{-1}(\tilde{\sigma}) = \uparrow 3 \downarrow 4$.

The paths used to construct EBAM are the same as those used to construct DAOCT. To represent several paths is used the same ideia of the DAOCT case. Hence, the q-th path $p_q := (u_q(1), u_q(2), \ldots, u_q(l_q))$, where q, $l_q \in \mathbb{Z}^+$.

The sequence of events for the EBAM is a sequence of all codified events that occurred in the observation. A path represents exactly one prodution cycle in the system and associated to it there is always a sequence of events. This sequence is formed of all the events that were recorded in the path and its definition can be seen in Definition 3.4.

Definition 3.4 (EBAM Event Sequence)
Let $p = (u(1), u(2), \ldots u(l))$ where $l \in \mathbb{Z}^+$ be a path, then, the sequence of events $\tilde{s}$ is

$$\tilde{s} := (u(2) - u(1), u(3) - u(2), \ldots, u(l) - u(l-1)).$$

Its length is the number of differences and $\tilde{s}[i]$ is its i-th element, where $i \leq l$, $i \in \mathbb{Z}^+$. In other words, a sequence of events is a sequence of event vectors.

Note that the event sequence has the length reduced in one compared to the path, since there is $l-1$ consecutive differences in it, where l is the length of the path used to compute the event sequence.

To represent several event sequences is used the same ideia of the paths. Hence, the i-th event sequence $\tilde{s}_i := (u_i(2) - u_i(1),\, u_i(3) - u_i(2),\, \ldots,\, u_i(j_i) - u_i(j_i - 1))$, where $i, j \in \mathbb{Z}^+$.

Consider now that there are $r \in \mathbb{Z}^+$ observed paths, represented as follows.

$$\begin{aligned} p_1 &= (u_1(1),\, u_1(2),\, \ldots,\, u_1(l_1)) \\ p_2 &= (u_2(1),\, u_2(2),\, \ldots,\, u_2(l_2)) \\ &\vdots \\ p_r &= (u_r(1),\, u_r(2),\, \ldots,\, u_r(l_r)) \end{aligned}$$

For each path is possible to construct an event sequence. Then, it is possible to represent each path by the initial I/O status and its event sequence, as can be seen as follows.

$$\begin{aligned} p_1 \Rightarrow \tilde{s}_1 &= (u_1(2) - u_1(1),\, u_1(3) - u_1(2),\, \ldots,\, u_1(l_1) - u_1(l_1 - 1)) \therefore (u_1(1),\, \tilde{s}_1) \\ p_2 \Rightarrow \tilde{s}_2 &= (u_2(2) - u_2(1),\, u_2(3) - u_2(2),\, \ldots,\, u_2(l_2) - u_2(l_2 - 1)) \therefore (u_2(1),\, \tilde{s}_2) \\ &\vdots \\ p_r \Rightarrow \tilde{s}_r &= (u_r(2) - u_r(1),\, u_r(3) - u_r(2),\, \ldots,\, u_r(l_r) - u_r(l_r - 1)) \therefore (u_r(1),\, \tilde{s}_r) \end{aligned}$$

As the DAOCT, the EBAM has the free parameter k that leads to a trade off between model size and accuracy. It can be interpreted as how many events are associated to the state, since the state records the last k events. This feature is implemented in the model by modifying the sequence of events as can be seen in Definition 3.5.

Definition 3.5 (Modified Sequence of Events)
Let $\tilde{s}_i$ be a sequence of events, u_i be I/O vectors, l_i be the length of u_i and k be the free parameter, then, the modified sequence of events $\tilde{s}_i^k$ is

$$\tilde{s}_i^k := (y_i(1),\, y_i(2),\, \ldots,\, y_i(l_i))$$

where:

$$y_i(j) := \begin{cases} (\tilde{s}_i[j-k+1],\, \ldots,\, \tilde{s}_i[l]), \text{ if } k \leq l \leq l_i,\ j \in \mathbb{Z}^+ \\ (\tilde{s}_i[1],\, \ldots,\, \tilde{s}_i[l]) \qquad\quad , \text{ if } l < k. \end{cases}$$

The elements of $\tilde{s}_i^k$ are called modified event vectors.

For $k = 1$ the sequence of events does not change, therefore, $\tilde{s}_1^1 = \tilde{s}$. From the modified sequences of events definition, it is straightforward that each vertex $y_i(j)$ stores the last $k-1$ events executed in the path if $j > k$ and the last $j-1$ events if $j < k$. Example 3.1 illustrates the construction of the modified sequence of events.

Example 3.1 (Construction of the Modified Sequence of Events)
Using the paths p_1, p_2 and p_3 of the motivating example in Section 3.1, let us construct the event sequence and the modified sequence of events for the free parameter $k = 1$ and $k = 2$:

$$p_1 = \left(\begin{bmatrix}0\\1\\0\\0\\0\\0\\0\end{bmatrix}, \uparrow 5 \uparrow 6, \begin{bmatrix}0\\1\\0\\0\\1\\1\\0\end{bmatrix}, \uparrow 3, \begin{bmatrix}0\\1\\1\\0\\1\\1\\0\end{bmatrix}, \uparrow 7 \downarrow 3 \downarrow 5, \begin{bmatrix}0\\1\\0\\0\\0\\1\\1\end{bmatrix}, \uparrow 4 \uparrow 5 \downarrow 7, \begin{bmatrix}0\\1\\0\\1\\1\\1\\0\end{bmatrix} \right)$$

$$p_2 = \left(\begin{bmatrix}0\\1\\0\\1\\1\\1\\0\end{bmatrix}, \downarrow 4, \begin{bmatrix}0\\1\\0\\0\\1\\1\\0\end{bmatrix}, \uparrow 1 \downarrow 6, \begin{bmatrix}1\\1\\0\\0\\1\\0\\0\end{bmatrix}, \uparrow 3, \begin{bmatrix}1\\1\\1\\0\\1\\0\\0\end{bmatrix}, \uparrow 7 \downarrow 3 \downarrow 5, \begin{bmatrix}1\\1\\0\\0\\0\\0\\1\end{bmatrix}, \downarrow 1, \begin{bmatrix}0\\1\\0\\0\\0\\0\\1\end{bmatrix}, \uparrow 4 \uparrow 5 \downarrow 7, \begin{bmatrix}0\\1\\0\\1\\1\\0\\0\end{bmatrix} \right)$$

$$p_3 = \left(\begin{bmatrix}0\\1\\0\\1\\1\\0\\0\end{bmatrix}, \downarrow 4, \begin{bmatrix}0\\1\\0\\0\\1\\0\\0\end{bmatrix}, \uparrow 3, \begin{bmatrix}0\\1\\1\\0\\1\\0\\0\end{bmatrix}, \uparrow 7 \downarrow 3 \downarrow 5, \begin{bmatrix}0\\1\\0\\0\\0\\0\\1\end{bmatrix}, \uparrow 4 \uparrow 5 \downarrow 7, \begin{bmatrix}0\\1\\0\\1\\1\\0\\0\end{bmatrix} \right)$$

The first step is to convert the path to the event sequence $\tilde{s}$. This can be done in two ways: (i) via the consecutives differences; (ii) via the codification function over the events, since the events are available in this case.

$$\tilde{s}_1 = \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} \right)$$

$$\tilde{s}_2 = \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} \right)$$

$$\tilde{s}_3 = \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} \right)$$

For the free parameter $k = 1$ the modified sequences of events do not change, however, for $k = 2$ they are different as can be seen as follows:

$$\tilde{s}_1^2 = \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & -1 \\ 0 & 0 \\ 0 & -1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1 & 0 \\ 0 & 1 \\ -1 & 1 \\ 0 & 0 \\ 0 & -1 \end{bmatrix} \right)$$

$$\tilde{s}_2^2 = \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \\ 0 & -1 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & -1 \\ 0 & 0 \\ 0 & -1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & -1 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & -1 \end{bmatrix} \right)$$

$$\tilde{s}_3^2 = \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & -1 \\ 0 & 0 \\ 0 & -1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1 & 0 \\ 0 & 1 \\ -1 & 1 \\ 0 & 0 \\ 1 & -1 \end{bmatrix} \right)$$

To obtain the original event vector from a modified event vector, it is used the event function ϕ, which definition can be seen in Definition 3.6.

Definition 3.6 (Event Function)
Given a modified event vector M, an I/O vector u and the free parameter k, the event function ϕ is defined as:

$$\begin{aligned} \phi : \mathbb{1}^{|u|_v \times k} &\to \mathbb{1}^{|u|_v} \\ M &\mapsto M[*, |u|_v] \end{aligned}$$

where $M[*, |u|_v]$ denotes the $|u|_v$-th column of M.

In addition, to obtain the event associated to an event vector, it is used the inverse of the codification function. For example, considering $e = \tilde{s}_3^2[4]$:

$$e = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1 & 0 \\ 0 & 1 \\ -1 & 1 \\ 0 & 0 \\ 1 & -1 \end{bmatrix} \Rightarrow \phi(e) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} \Rightarrow \sigma = f_c^{-1}(\phi(e)) = \uparrow 4 \uparrow 5 \downarrow 7.$$

The last definition is the EBAM path, that is the path used to construct the model and can be seen in Definition 3.7.

Definition 3.7 (EBAM path)
An EBAM path $\tilde{p}$ is a tuple formed by the initial I/O status, as first element, and the modified sequence of events, as the others elements, of a given path p. Its length $|\tilde{p}|_p$ is the same of p, then, $|\tilde{p}|_p = |p|_p$.

The EBAM is obtained following the steps of Algorithm 1.

Algorithm 1: EBAM Construction

Input: EBAM paths $(u_i(1), \tilde{s}_i)$, $\forall i \in \{1, 2, \ldots, r\}$, $r \in \mathbb{Z}^+$

Output: $EBAM = (X, x_0, \Sigma, \delta, \lambda, \lambda_0, \Omega, \theta)$

$\Omega \leftarrow \{1, 2, \ldots, r\}$ // *Defining the set of path indexes*
Create the state x_0
$X \leftarrow \{x_0\}$ // *Creating the set of states*
$\Sigma \leftarrow \lambda_0(x_0) \leftarrow \{\ \}$ // *Creating the set of events and the initial state label*
for $i \in \Omega$ **do**
    for $j \in \{1, 2, \ldots, |\tilde{s}_i|_s\}$ **do**
        if $j = 1$ **then**
            $x_c \leftarrow x_0$ // *Updating the current state*
            $\lambda_0(x_0) \leftarrow \lambda_0(x_0) \cup \{u_i(1)\}$ // *Updating the initial state label*
        else
            $x_p \leftarrow x_c$ // *Updating the previous state*
            $\sigma \leftarrow f_c^{-1}(\phi(\tilde{s}_i[j]))$ // *Event*
            if $\sigma \in \Sigma$ **then**
                $x_c \leftarrow \delta(x_p, \sigma)$ // *Updating the current state*
            else
                Create the state $x_{\|X\|}$
                $x_c \leftarrow x_{\|X\|}$ // *Updating the current state*
                $X \leftarrow X \cup \{x_c\}$ // *Updating the set of states*
                $\Sigma \leftarrow \Sigma \cup \{\sigma\}$ // *Updating the set of events*
                $\lambda(x_c) \leftarrow \tilde{s}_i[j]$ // *Defining the state output*
            if $\delta(x_p, \sigma)!$ **then**
                $\theta(x_p, \sigma) \leftarrow \theta(x_p, \sigma) \cup \{i\}$ // *Updating the allowed index path*
            else
                $\delta(x_p, \sigma) \leftarrow x_c$ // *Creating the transition*
                $\theta(x_p, \sigma) \leftarrow \{i\}$ // *Defining the allowed index path*

In order to elucidate the Algorithm 1, it will be explained line by line. The inputs are the observed paths and the output is the EBAM. Lines 1 to 4 create the sets Ω, X, Σ and $\lambda_0(x_0)$, that will be filled throughout the algorithm, and the initial state x_0. The *for* blocks in Lines 5 and 6 run the algorithm for every modified event inside each EBAM path. The *if* block in Line 7 is just to pick the first modified event of a path. Inside of it, the current state now is defined as the initial one and the initial state label is updated. The *else* block in Line 10 picks the rest of the modified events and defines the previous state as the current one and converts the modified event to the real event. The *if* block in Line 13 checks if the event is already in the model. If it is, the current state is updated to following the transition based on the δ function. If it is not the first time that this event appears, represented in Line 15 in the *else* block, then, a new state is created and it is defined as the current one. Besides, the set of states and set of events are updated and the state output for the current state is defined. The following *if* block, in Line 21, verifies if the transition from the previous state with the event is defined. If it is, then, the allowed index path function is updated, that is, the index of the current path is added to the possible indexes to this transition. If the transition is not defined, represented in the *else* block, in Line 23, then, the transition is created and the allowed index path function is defined for this transition as the set containing only the index of the current path.

To illustrate the construction of the EBAM, an example will be given in Example 3.2.

Example 3.2 (EBAM Construction)
Let us construct the EBAM of the motivating example in Section 3.1 for the free parameter $k = 1$ and $k = 2$.

For the free parameter $k = 1$, the constructed model can be seen in Figure 3.2.

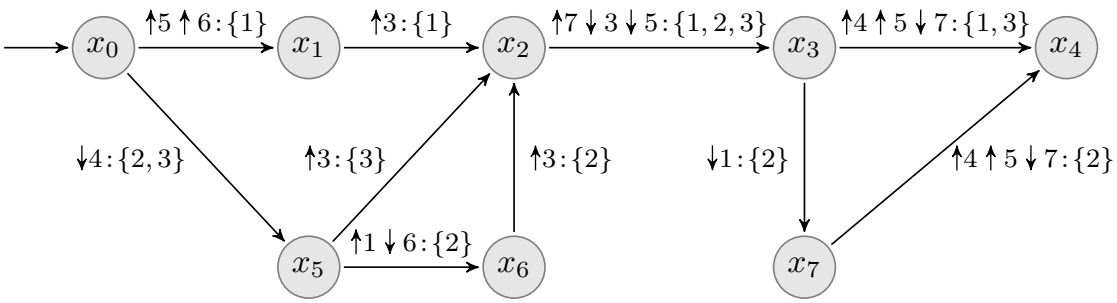

Figure 3.2: EBAM for $k = 1$ of the Example 3.2.

And for the free parameter $k = 2$, the constructed model can be seen in Figure 3.3.

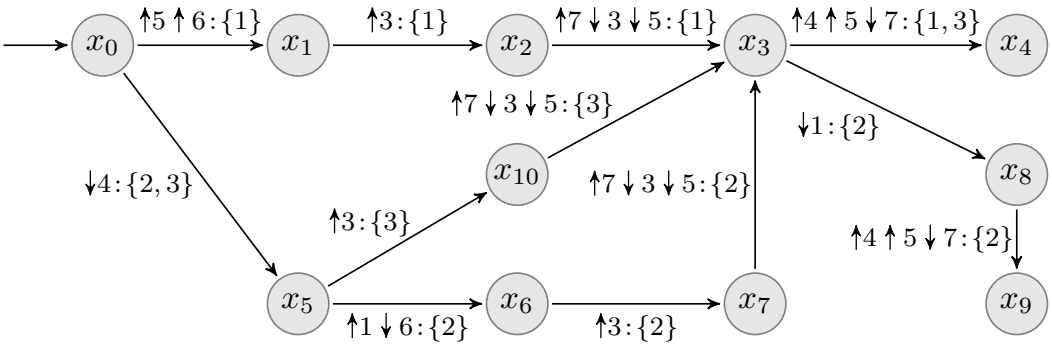

Figure 3.3: EBAM for $k = 2$ of the Example 3.2.

The EBAM of the Figure 3.2, has the following attributes:

- $X = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$;
- $x_0 = x_0$ (It is important to highlight that x_0 is commonly used as the attribute of the model and as the name of the initial state);
- $\Sigma = \{\downarrow 1, \uparrow 3, \downarrow 4, \uparrow 1 \downarrow 6, \uparrow 5 \uparrow 6, \uparrow 4 \uparrow 5 \downarrow 7, \uparrow 7 \downarrow 3 \downarrow 5\}$;
- $\delta(x_0, \uparrow 5 \uparrow 6) = x_1$, $\delta(x_0, \downarrow 4) = x_5$, $\delta(x_1, \uparrow 3) = x_2$, $\delta(x_5, \uparrow 3) = x_2$, $\delta(x_5, \uparrow 1 \downarrow 6) = x_6$, $\delta(x_6, \uparrow 3) = x_2$, $\delta(x_2, \uparrow 7 \downarrow 3 \downarrow 5) = x_3$, $\delta(x_3, \uparrow 4 \uparrow 5 \downarrow 7) = x_4$, $\delta(x_3, \downarrow 1) = x_7$, $\delta(x_7, \uparrow 4 \uparrow 5 \downarrow 7) = x_4$;
- $\lambda_0(x_0) = \left\{ \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \right\}$;
- $\lambda(x_1) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$, $\lambda(x_2) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $\lambda(x_3) = \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}$, $\lambda(x_4) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix}$, $\lambda(x_5) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$,

$$\lambda(x_6) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ 0 \end{bmatrix}, \lambda(x_7) = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix};$$

- $\Omega = \{1, 2, 3\}$;
- $\theta(x_0, \uparrow 5 \uparrow 6) = \{1\}$, $\theta(x_0, \downarrow 4) = \{2, 3\}$, $\theta(x_1, \uparrow 3) = \{1\}$, $\theta(x_5, \uparrow 3) = \{3\}$, $\theta(x_5, \uparrow 1 \downarrow 6) = \{2\}$, $\theta(x_6, \uparrow 3) = \{2\}$, $\theta(x_2, \uparrow 7 \downarrow 3 \downarrow 5) = \{1, 2, 3\}$, $\theta(x_3, \uparrow 4 \uparrow 5 \downarrow 7) = \{1, 3\}$, $\theta(x_3, \downarrow 1) = \{2\}$, $\theta(x_7, \uparrow 4 \uparrow 5 \downarrow 7) = \{2\}$.

The EBAM of the Figure 3.3, has the following attributes:

- $X = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}\}$;
- $x_0 = x_0$ (It is important to highlight that x_0 is commonly used as the attribute of the model and as the name of the initial state);
- $\Sigma = \{\downarrow 1, \uparrow 3, \downarrow 4, \uparrow 1 \downarrow 6, \uparrow 5 \uparrow 6, \uparrow 4 \uparrow 5 \downarrow 7, \uparrow 7 \downarrow 3 \downarrow 5\}$;
- $\delta(x_0, \uparrow 5 \uparrow 6) = x_1$, $\delta(x_0, \downarrow 4) = x_5$, $\delta(x_1, \uparrow 3) = x_2$, $\delta(x_2, \uparrow 7 \downarrow 3 \downarrow 5) = x_3$, $\delta(x_5, \uparrow 3) = x_{10}$, $\delta(x_{10}, \uparrow 7 \downarrow 3 \downarrow 5) = x_3$, $\delta(x_5, \uparrow 1 \downarrow 6) = x_6$, $\delta(x_6, \uparrow 3) = x_7$, $\delta(x_7, \uparrow 7 \downarrow 3 \downarrow 5) = x_3$, $\delta(x_3, \uparrow 4 \uparrow 5 \downarrow 7) = x_4$, $\delta(x_3, \downarrow 1) = x_8$, $\delta(x_8, \uparrow 4 \uparrow 5 \downarrow 7) = x_9$;
- $\lambda_0(x_0) = \left\{ \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \right\}$;
- $\lambda(x_1) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \lambda(x_2) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}, \lambda(x_3) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & -1 \\ 0 & 0 \\ 0 & -1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix},$

$$\lambda(x_4) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 0 & 0 \\ 1 & -1 \end{bmatrix}, \lambda(x_5) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \lambda(x_6) = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \\ 0 & -1 \\ 0 & 0 \end{bmatrix},$$

$$\lambda(x_7) = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \end{bmatrix}, \lambda(x_8) = \begin{bmatrix} 0 & -1 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}, \lambda(x_9) = \begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \end{bmatrix},$$

$$\lambda(x_{10}) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ -1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix};$$

- $\Omega = \{1, 2, 3\}$;
- $\theta(x_0, \uparrow 5 \uparrow 6) = \{1\}$, $\theta(x_0, \downarrow 4) = \{2, 3\}$, $\theta(x_1, \uparrow 3) = \{1\}$, $\theta(x_2, \uparrow 7 \downarrow 3 \downarrow 5) = \{1\}$, $\theta(x_5, \uparrow 3) = \{3\}$, $\theta(x_{10}, \uparrow 7 \downarrow 3 \downarrow 5) = \{3\}$, $\theta(x_5, \uparrow 1 \downarrow 6) = \{2\}$, $\theta(x_6, \uparrow 3) = \{2\}$, $\theta(x_7, \uparrow 7 \downarrow 3 \downarrow 5) = \{2\}$, $\theta(x_3, \uparrow 4 \uparrow 5 \downarrow 7) = \{1, 3\}$, $\theta(x_3, \downarrow 1) = \{2\}$, $\theta(x_8, \uparrow 4 \uparrow 5 \downarrow 7) = \{2\}$.

3.3 Languages

EBAM is obtained by identification, therefore, the system is observed and the data is recorded. However, without knowing the system behavior, it is impossible to know when the behavior is completely recorded. Theoretically, only with infinite observation this can be assumed to be true. Hence, the observed behavior can be expected to be a subset of the original fault-free behavior. In order to reduce the difference between these sets, the observation is, in general, carried out for a long time. There are six important languages and their relations are shown in Figure 3.4.

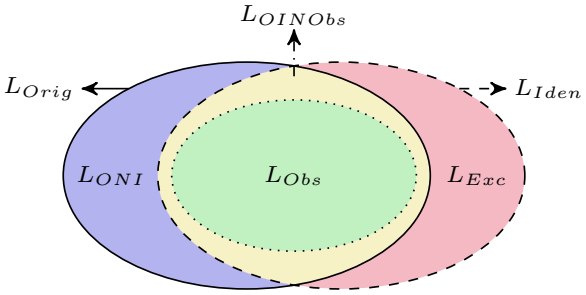

Figure 3.4: The relation between every language in the EBAM.

In Figure 3.4, the languages are:

- L_{Orig} is the original language of the system, *i.e.*, the language generated by the fault-free behavior;
- L_{Obs} is the observed language;
- L_{Iden} is the identified language;
- L_{ONI} is the language that is original but not identified;
- L_{Exc} is the exceeding language, *i.e.*, part of identified language that is not part of the original one;
- L_{OINObs} is the original language that is also identified, however it has not been observed.

The language observed used to generate by the EBAM is $L_{Obs} := \bigcup_{i=1}^{r} \bar{\tilde{s}}_i$.

To define the identified language, we need, before, to define some concepts as the verification function and feasible states from an I/O vector. The verification function f_v can be defined as follows:

$$f_v : \mathbb{Z}_2^n \times \mathbb{1}^n \to \mathbb{Z}_2$$

$$f_v(u, \tilde{\sigma}) := \begin{cases} 1, & \text{if } (u + \tilde{\sigma}) \in \mathbb{1}^n \\ 0, & \text{otherwise.} \end{cases}$$

The verification function returns 0 if any element of the vector $u + \tilde{\sigma}$ is not -1 nor 0 nor 1, otherwise it returns 1. In words, this function checks if the new vector (associated to a next state) makes sense for the model.

It is possible to extend the verification function in a natural way as follows:

$$f_v : \mathbb{Z}_2^n \times (\mathbb{1}^n)^k \to \mathbb{Z}_2$$

$$f_v(u, \tilde{s}\tilde{\sigma}) := f_v(f_v(u, \tilde{s}), \tilde{\sigma})$$

Now it is possible to define the feasible language generated by one I/O vector as follows:

$$l_{feas}(u) := \{\tilde{s} \in (\mathbb{1}^n)^k, k \in \mathbb{N} \mid (\delta(x_0, s)!, s = f_c^{-1}(\tilde{s})) \wedge (f_v(u, t) = 1, \forall t \in \bar{\tilde{s}})\}.$$

Therefore, the EBAM feasible language is the union of all feasible languages that start from the initial states, then:

$$L_{feas} := \bigcup_{u_0 \in \lambda_0(x_0)} l_{feas}(u_0)$$

To proceed to the identified language, it is necessary to extend the transition function δ to also allow sequences instead of only events, then:

$$\delta : X \times \Sigma^k \to X$$

$$\delta(x, \sigma_0\sigma_1) := \delta(\delta(x, \sigma_0), \sigma_1)$$

Now, instead of expanding the path estimation function θ to allow sequences, a new function θ_s is recursively defined as follows:

$$\theta_s : \lambda_0(x_0) \times L_{feas} \to 2^{\Omega}$$

$$\theta_s(u, \varepsilon) := \{i \in \Omega \mid u_i(1) = u\}$$

$$\theta_s(u, \tilde{s}\tilde{\sigma}) := \theta_s(u, \tilde{s}) \cap \theta(x, \tilde{\sigma}), \text{ where } \sigma = f_c^{-1}(\tilde{\sigma}),\ x = \delta(x_0, s),\ s = f_c^{-1}(\tilde{s})$$

Now it is possible to define the identified language by the EBAM as follows:

$$L_{Iden} := \{\tilde{s} \in L_{feas} \mid (\exists u_0 \in \lambda_0(x_0)) \wedge (\theta_s(u_0, \tilde{s}) \neq \emptyset)\}.$$

The language formed of all subwords of events of length up to n generated by the EBAM: $L_{Iden_S}^{\leq n} := \{\tilde{s} \in (\mathbb{Z}_2^n)^* : \tilde{s} \in L_{Iden} \wedge |\tilde{s}|_s \leq n\}$.

The language original but not identified L_{ONI} is defined as $L_{ONI} := L_{Orig} \setminus L_{Iden}$.

The language formed of all subwords of events of length up to n generated by the sequences of a language L_l: $L_{l_S}^{\leq n} := \{\tilde{s} \in (\mathbb{Z}_2^n)^* : \tilde{s} \in L_l \wedge |\tilde{s}|_s \leq n\}$.

The exceeding language L_{Exc} is defined as $L_{Exc} := L_{Iden} \setminus L_{Orig}$.

The last language is the original language that is also identified but not observed L_{OINObs}, that is defined as $L_{OINObs} := (L_{Orig} \cap L_{Iden}) \setminus L_{Orig}$.

However, as shown in [23], if a sufficiently large number of I/O status are observed, then, there exists a number $n_0 \in \mathbb{Z}^+$ such that $L_{\overline{Orig}_S}^{\leq n_0} \setminus L_{\overline{Obs}_S}^{\leq n_0} \approx 0$. In words, this means that it is assumed that all the fault-free behavior of the system was observed. Therefore, $L_{\overline{ONI}_S}^{\leq n_0} \approx 0$, since $L_{Obs} \subseteq L_{Orig}$ and $L_{Obs} \subseteq L_{Iden}$. In addition, in this work, it is assumed that $L_{\overline{ONI}_S}^{\leq n_0} = 0$ in order to reduce the occurence of false alarms. It is important to remark that the fault detection can be performed even without this assumption, however, there is a higher risk of raising false alarms.

Considering $L_{\overline{Orig}_S}^{\leq n_0} \setminus L_{\overline{Obs}_S}^{\leq n_0} \approx 0$, then, the exceeding language L_{Exc} is now also defined as $L_{Exc} := L_{Iden} \setminus L_{Obs}$ and the original language that is also identified but not observed L_{OINObs} is now also defined as $L_{OINObs} := (L_{Orig} \cap L_{Iden}) \setminus L_{Obs}$.

3.4 Properties

EBAM and DAOCT share several properties as simulate the observed fault-free language of the system, k-completeness and no exceeding language if there is no cyclic paths for a given value of k. In addition, EBAM is, in general, more compact than DAOCT for a given value of k. The EBAM properties are theorems and will be demonstrated in the sequel.

Theorem 3.1 (Simulation)
EBAM simulates the observed fault-free language of the system, i.e., $L_{Obs} \subseteq L_{Iden}$.

Proof. Let $\tilde{s}_i = \tilde{\sigma}_i(1)\tilde{\sigma}_i(2)\ldots\tilde{\sigma}_i(l_i - 1)$ be a modified sequence of events associated to an EBAM path $\tilde{p}_i = (u_i(1), \tilde{s}_i)$, where $i \in \mathbb{Z}^+$ and $l_i = |\tilde{p}_i|_p$. According to Algorithm 1, there is a path in the EBAM $(x_1, \tilde{\sigma}_i(1), x_2, \tilde{\sigma}_i(2), \ldots, \tilde{\sigma}_i(l_i - 1), x_{l_i})$, associated with $\tilde{p}_i$, where x_i is not necessarily distinct from x_j, for $i, j = 1, 2, \ldots, l_i$ such that $i \neq j$ and for $m = 1, 2, \ldots, l_i - 1$ such that $i \in \theta(x_m, \tilde{\sigma}_i(m))$. Thus, any prefix of s belongs to the language generated by EBAM, which implies that $L_{Obs} \subseteq L_{Iden}$. □

Theorem 3.2 (k-completeness)
For a given value of the free parameter $k \in \mathbb{Z}^+$, the EBAM is k-complete, i.e., $L_{\overline{Iden}_S}^{\leq n} = L_{\overline{Obs}_S}^{\leq n}$, for all $n \leq k$.

Proof. Since according to Theorem 3.1, $L_{Obs} \subseteq L_{Iden}$, then, $L_{\overline{Obs}_S}^{\leq n} \subseteq L_{\overline{Iden}_S}^{\leq n}$, for all $n \leq k$. Let us now prove that $L_{\overline{Iden}_S}^{n} \supseteq L_{\overline{Obs}_S}^{\leq n}$, Let $p = (x_1, \tilde{\sigma}(1), x_2, \tilde{\sigma}(2), \ldots, \tilde{\sigma}(n), x_{n+1})$ be a feasible path of the EBAM of length $n+1$,

i.e., $\theta_s(x_1, \tilde{\sigma}(1)\tilde{\sigma}(2)\ldots\tilde{\sigma}(n)) \neq \emptyset$. According to Algorithm 1, any transition of p is associated with a transition in at least one path $\tilde{p}_i$, for $i = 1, 2, \ldots, r$, where $r \in \mathbb{Z}^+$. Let us consider the last transition of p, *i.e.*, $(x_n, \tilde{\sigma}(n), x_{n+1})$, and assume that $(\lambda(x_n), \sigma_n, \lambda(x_{n+1})$ is the transition in path $\tilde{p}_i$ where $i \in \{1, 2, \ldots, r\}$, associated with transition $(x_n, \tilde{\sigma}(n), x_{n+1})$. Let $\lambda(x_n) = y_i(n)$. Since all suffixes of length $1, 2, \ldots, k-1$ of the sequences that reach $y_x(n)$ must also belong to $\tilde{p}_i$. Consequently, $\tilde{\sigma}(1)\tilde{\sigma}(2)\ldots\tilde{\sigma}(n) \in L^{\leq n}_{Obs_S}$, for all $n \leq k$, which implies that $L^{\leq n}_{Iden_S} \subseteq L^{\leq n}_{Obs_S}$, for $n \leq k$. □

Theorem 3.3 (Exceeding Language)
If the EBAM does not have cyclic paths for a given value of the free parameter $k \in \mathbb{Z}^+$, then, there is no exceeding language, i.e., $L_{Exc} = \emptyset$.

Proof. Notice, according to Algorithm 1, that each transition of the EBAM is associated with at least one observed EBAM path $p_i = (u_i(1), \tilde{s}_i)$, where $i = 1, 2, \ldots, r$ and $r \in \mathbb{Z}^+$ and u is an I/O vector. Moreover, since all events of the path p_i are observale, then, associated with each path p_i there is a number $n_i < l_i$, where $l_i \in \mathbb{Z}^+$ such that p_i can be distinguished from all paths after the observation of n_i events. Consequently, since the EBAM does not have cyclic paths, then, after the occurance of the observed modified sequence of events $\tilde{s}_i$, we have that $\theta_s(x_0, \tilde{s}_i) = \{i\}$. In addition, since the EBAM is acyclic, the intersection of the path estimates of two transitions leaving the same state of the EBAM must be empty, which implies that all paths p_i are uniquely determined before reaching its corresponding final state. Thus, if a modified sequence $\tilde{s}$ associated to a sequence $s \in \Sigma^* \setminus L_{Obs}$ is observed, the two possibilities may happen: (i) $\delta(x_0, \tilde{s})$ is not defined; (ii) $\delta(x_0, \tilde{s})$ is defined, however, in this case, $\theta_s(x_0, \tilde{s}) = \emptyset$. Thus, $\tilde{s} \notin L_{Iden}$, which implies that $L_{Iden} \subseteq L_{Obs}$, and, therefore, $L_{Exc} = \emptyset$. □

3.5 Fault Detection

One of the fault diagnosis based on the identified EBAM goals is to diagnose faults in real time, *i.e.*, be able to read data from the system online, compare it with the model prediction and evaluate if a fault occurred. In order to do it, a scheme to diagnose a fault using the EBAM, which works online and offline, is proposed and it is inspired by the one presented in [14, 29], where conditions are defined to indicate if the observed sequence of events does not correspond to the modeled fault-free behavior of the system. This scheme can be seen in Figure 3.5 and is proposed as follows:

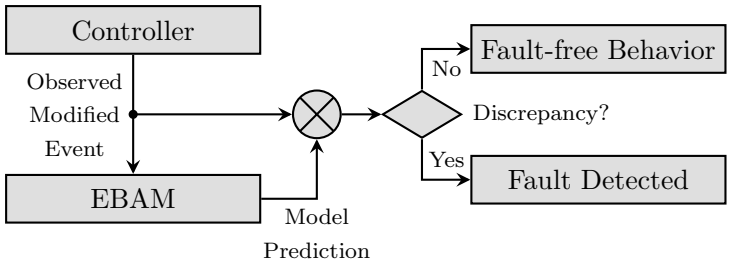

Figure 3.5: Fault detection scheme based on the EBAM model.

1) Syncronize the system and the model;
The system must start in one of the possible initial states of the paths (since it is allowed to have more than one) and the EBAM must start at the initial state (x_0).

2) Observe the system;
Observe every event from the system as it evolves.

3) Compute the modified event for every event;
Since the EBAM uses modified events instead of real events, it is necessary to convert the events.

4) Play the model.
Check if the modified event is possible at the current state of the EBAM.

- If the modified event is allowed, evaluate the next EBAM state;
  - If it is a final state of a path and the number of observed events from the initial state is equal to the length of this path, its state label must also be checked;
    - If it is in the state label of the initial state, the EBAM and the event counter must be restarted and the scheme returns to Step **2**.
    - Otherwise, a fault is detected (the next sequence of events were not observed in the paths) and the scheme is finished.
  - Otherwise, if the state is not a final state of any path and the number of events is sufficient to distinguish paths, the behavior is fault-free, then, there is no fault and the scheme returns to Step **2**.
- Otherwise, if the modified event is not allowed and the number of events is sufficient to distinguish paths, a fault is detected and the scheme is finished.

Two important concepts are the total number of events since the model started (until it restarts) and the number of events to distinguish paths. The first one is important because if a path has length $l \in \mathbb{Z}^+$, it is expected that this path reaches its final vertex after $l-1$ observations. If it reaches after a different number of observations, then, a fault has occurred, since the path was not executed. In order to obtain the number of events in an EBAM path, function π is introduced in Definition 3.8

Definition 3.8 (Number of Events Function)
Let Ω be the finite set of path indexes and $\tilde{p}_i$, for $i \in \Omega$, be EBAM paths, then, the number of events function is defined as follows:

$$\begin{aligned} \pi : \Omega &\to \mathbb{Z}^+ \\ i &\mapsto |\tilde{p}_i|_p - 1 \end{aligned}$$

Other important function is the last I/O status function. It evaluates the last I/O status from an EBAM path and it can be seen in Definition 3.9.

Definition 3.9 (Last I/O Status Function)
Let Ω be the finite set of path indexes, $\tilde{p}_i$, for $i \in \Omega$ be EBAM paths and u be the I/O vector associated to $\tilde{p}_i$, then, the last I/O status function is defined as follows:

$$\begin{aligned} \psi : \Omega &\to \mathbb{1}^{|u|_v} \\ i &\mapsto \sum_{j=1}^{|\tilde{p}_i|_p} \tilde{p}_i[j] \end{aligned}$$

The next concept is the number of events to distinguish paths. The main idea behind this concept is to be able to identify which path is playing in the model. Until the path is determined, it is not possible to ensure that a fault has occurred. For this reason, each path has associated to itself a number, that represents the minimum number of observations to be sure that the path is determined. Since it is considered that the sequence s_a of a path $\tilde{p}_a$ cannot be a prefix of a different sequence s_b of another path $\tilde{p}_b$, for $a, b \in \Omega$ that starts with the same I/O vector, then, there always exists a positive integer number $n_a \leq \pi(a)$, associated with s_a, such that the observation of the prefix of s_a with length n_a is sufficient to distinguish the path $\tilde{p}_a$ from all other paths that start with the same I/O vector as $\tilde{p}_b$. This concept is defined as a function and can be seen in Definition 3.10.

Definition 3.10 (Minimum Number of Events Function)
Let Ω be the finite set of path indexes and $\tilde{p}_i$, for $i \in \Omega$ be EBAM paths, then, the minimum number of events function is defined as follows:

$$\omega : \Omega \to \mathbb{Z}^+ \cup \{0\}$$
$$i \mapsto \omega(i)$$

$$\omega(i) := \min(\{l - 1 : \tilde{p}_i[1:l] \neq \tilde{p}_j[1:l],\ \forall j \in \Omega \setminus \{i\},\ \forall l \in \{1, 2, \ldots, |\tilde{p}_i|_p\}\})$$

Where $\tilde{p}_i[a : b]$, for $a, b \in \{1, 2, \ldots, |\tilde{p}_i|_p\}$ and $b \geq a$, denotes the tuple $(\tilde{p}_i[a], \ldots, \tilde{p}_i[b])$.

The increase in the value of the free parameter k reduce cycles in the model, that implies in a reduction of the exceeding language. Another function that also avoids cycles, is the function that controls the allowed path indexes. Associated to each transition in the model, it may avoid cycles during the execution of a path. It is important to remind that in this case, the structural cycle will be in the model, but a path will not be allowed to go through this type of cycle.

There are 3 types of cycles: (i) the cycles allowed in the model and feasible in the system; (ii) the cycles allowed in the model and not feasible in the system and (iii) structural cycles that are not possible in the model due to the path estimation function.

The Example 3.3 illustrates the concepts discussed above.

Example 3.3 (Example of Basic Concepts)
Let u be an I/O vector such that $|u|_v = 4$ and $\tilde{p}_1$, $\tilde{p}_2$ and $\tilde{p}_3$ be EBAM paths generated by u.

$$\begin{cases} \tilde{p}_1 = ((3,4),\ \downarrow 4,\ \downarrow 3,\ \uparrow 1,\ \downarrow 1) \\ \tilde{p}_2 = ((),\ \uparrow 1,\ \downarrow 1,\ \uparrow 2,\ \uparrow 3,\ \uparrow 4, \downarrow 2 \downarrow 4,\ \uparrow 4,\ \downarrow 4) \\ \tilde{p}_3 = ((3),\ \uparrow 1,\ \downarrow 1,\ \uparrow 2,\ \downarrow 3) \end{cases}$$

To simplify visualization, the following definitions are made:

$$\sigma_1 = \downarrow 4,\ \sigma_2 = \downarrow 3,\ \sigma_3 = \uparrow 1,\ \sigma_4 = \downarrow 1,\ \sigma_5 = \uparrow 2,\ \sigma_6 = \uparrow 3,\ \sigma_7 = \uparrow 4,\ \sigma_8 = \downarrow 2 \downarrow 4$$

Replacing the events values in $\tilde{p}_i$ for $i = 1, 2, 3$, the representations are now as follows:

$$\begin{cases} \tilde{p}_1 = ((3,4),\ \sigma_1,\ \sigma_2,\ \sigma_3,\ \sigma_4) \\ \tilde{p}_2 = ((),\ \sigma_3,\ \sigma_4,\ \sigma_5,\ \sigma_6,\ \sigma_7,\ \sigma_8,\ \sigma_7,\ \sigma_1) \\ \tilde{p}_3 = ((3),\ \sigma_3,\ \sigma_4,\ \sigma_5,\ \sigma_2) \end{cases}$$

Using Algorithm 1 for these paths, the models for $k = 1, 2$ can be seen in Figures 3.6 and 3.7, respectively.

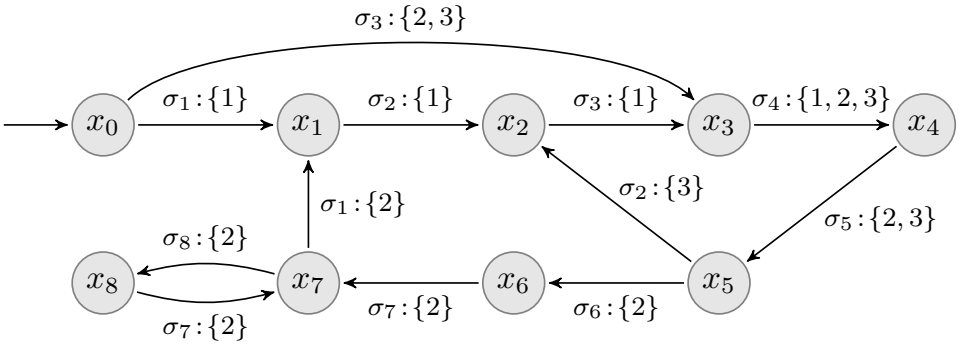

Figure 3.6: EBAM for $k = 1$ of the Example 3.3.

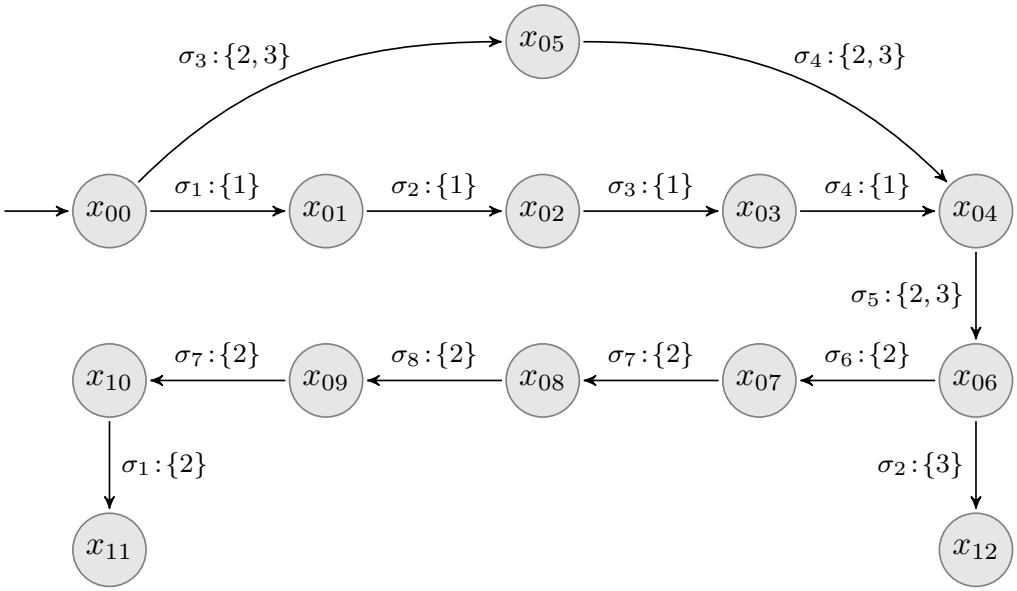

Figure 3.7: EBAM for $k = 2$ of the Example 3.3.

Let $s = \sigma_3\sigma_4\sigma_5\sigma_6\sigma_7\sigma_1$. Playing this sequence of events in the model for $k = 1$ (Figure 3.6), the last state will be $\delta(x_0, s) = x_1$ and this means that path 2 is the correct one, since $\theta_s(x_0, s) = \{2\}$. In addition, note that state x_1 is the last state of the path $\tilde{p}_2$. Thus, one may think that it is a valid play, however, if the number of events is counted, a problem appears, since $|\tilde{p}_2|_p - 1 = 9 \neq |s|_s = 6$. Therefore, s does not represent path 2 and it is a faulty sequence of events. This shows the importance of counting the events.

Comparing the models for $k = 1, 2$, one can note that while the model for $k = 1$ has several cycles, the model for $k = 2$ is cycle-free. Futhermore, for $k = 1$, two types of cycles are present ((ii) and (iii)).

The first cycle is a cycle of type (ii), *i.e.*, it is the structural one that allows the model to play forever but it is not physically feasible. In state x_7, it is possible to play forever the sequence of events $\sigma_8\sigma_7$. However, looking carefully the events, one can note that it is physically impossible to happen, since $\sigma_7 = \uparrow 4$ and $\sigma_8 = \downarrow 2 \downarrow 4$ and after σ_8, $u[2] = 0$. Then, after σ_7, $u[2]$ will not change and this is the problem, once, $u[2] = 0$, there is no way in the real system to event σ_8 happen again.

The second cycle is a cycle of type (iii), *i.e.*, it is the structural one that is not possible to be executed. In state x_2, the sequence of events $\sigma_3\sigma_4\sigma_5\sigma_2$ closes a loop, however, this is not a valid sequence, since the θ function differs the paths. Event σ_3 is only possible in state x_2 if the index path is 1, otherwise, σ_3 is not allowed in state x_4, therefore, it is just a structural cycle. The model for $k = 2$ does not have any cycles and it shows how the free parameter k can help avoiding cycles.

It is important to remark that the EBAM must be reinitializable in order to be used for fault detection, since EBAM may represent several paths and the last I/O status, after a playing, becomes the initial one for the next path. This concept was first introduced in [14] and the definition of model reinitializability for the EBAM case is presented in the sequel.

Definition 3.11 (EBAM Reinitializability)
Let $\tilde{s}_i$ be the sequence of events associated to the EBAM path $\tilde{p}_i$, for $i \in \Omega$. Then, the EBAM is said to be reinitializable if $\nexists \tilde{s}' \in \overline{\{\tilde{s}_i\}}$ of length $|\tilde{s}'|_s = \pi(j)$, where $j \in \theta_s(\tilde{p}_i[1], \tilde{s}')$ and $\pi(j) < \pi(i)$, such that $\psi(j) = \tilde{p}_i[1] + \sum_{q=1}^{\pi(j)-1} \tilde{s}_i[q]$.

In words, the EBAM, which initial I/O status is u_0, is reinitializable when after a sequence of events s with length $|s|_s = \pi(j)$ where $j \in \Omega$, the actual I/O status $u(\pi(j))$ is the final I/O status of the path p_j, u_n it is an initial I/O status of at least one path and j is a valid index path, *i.e.*, $j \in \theta_s(u_0, s)$.

As shown in Figure 3.5, during fault detection, the system is observed and the observed events are converted to vectors and, then, played in the model. In this context, an event is said to be viable if it is playable in the model at the current state. When a event is viable, this means that the actual sequence of events is fault-free, *i.e.*, it was observed during the observation of the fault-free behavior of the system. There are some conditions for an event to be considered viable and them can be seen in Definition 3.12.

Definition 3.12 (EBAM Fault Detection Conditions)
Let $s \in \Sigma^*$ be the previous observed sequence of events generated by the system, u be the I/O vector and u_0 the initial I/O status observed such that $\theta(u_0, s) \neq \emptyset$. Let $x_c = \delta(x_0, s)$ be the current state, σ be the next observed event and $u(n)$ be the I/O status after the occurrence of sequence $s\sigma$. Then, σ is said to be viable in the current state $x_c \in X$ of the EBAM if the following conditions hold true:

C1. $\sigma \in \gamma(x_c)$;

C2. $\theta_s(u_0, s\sigma) \neq \emptyset$;

C3. If $\|\theta_s(u_0, s)\| > 1$ and $\theta_s(u_0, s\sigma) = \{i\}$, for $i \in \Omega$, then, $|s\sigma|_s \geq \omega(i)$;

C4. If $|s\sigma|_s = \pi(i)$, for $i \in \theta_s(u_0, s\sigma)$, then, $u(n) = \psi(i)$ or $\exists j \in \theta_s(u_0, s\sigma)$ such that $|s\sigma|_s < \pi(j)$;

C5. If $|s\sigma|_s = \pi(i)$ and $u(n) = \psi(i)$, for $i \in \theta_s(u_0, s\sigma)$, then, $u(n) \in \lambda_0(x_0)$.

According to the EBAM identified language, a sequence of events $s\sigma$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$, belongs to the identified language L_{Iden} if, and only if, conditions **C1** and **C2** are satisfied. If condition **C3** is not true, then, the path $\tilde{p}_i$, where $i \in \Omega$, can be distinguished from all other paths that start with the same I/O status by observing the prefix of s_i with length smaller than $\pi(i)$, which is not possible in the fault-free system behavior. If condition **C4** is not verified, then, it is not possible to reach the final I/O status $\psi(i)$ of a path p_i such that $i \in \theta_s(u, s\sigma)$, after the observation of $s\sigma$, which implies that a fault has occurred. Finally, if condition **C5** is not verified, then, the model after being reinitialized will be in an I/O status that is not in the initial state label, *i.e.*, it is not the initial state for any path of the model, which avoids the model to be reinitialized. Thus, the fault detection is carried out by verifying if the observed event is viable in the current state of the fault-free system model. If the observed event is not viable, then, the fault is detected.

With the conditions presented in Definition 3.12, the scheme can be formalized in an algorithm and it is shown in Algorithm 2.

Algorithm 2: EBAM Fault Detection

Input: An EBAM

Output: Fault detection

Synchronize the system and the model (same fault-free initial state)
$u_p \leftarrow$ Current I/O status read from the system
$x_c \leftarrow x_0$ // *Defining the current state*
$\Theta_c \leftarrow \{i \in \Omega : \tilde{p}_i[1] = u_p\}$ // *Path estimation*
$e\_c \leftarrow 0$ // *Defining the event observations counter*
while True **do**
    Wait the next I/O status observation u_c such that $u_c \neq u_p$
    $\sigma \leftarrow f_c^{-1}(u_c - u_p)$ // *Event*
    $e\_c \leftarrow e\_c + 1$
    if $\sigma \notin \gamma(x_c)$ **then**
        Stop the algorithm, since a fault was detected
    $\Theta_c \leftarrow \Theta_c \cap \theta(x_c, \sigma)$ // *Updating the allowed index path*
    for $path \in \Theta_c$ **do**
        if $e\_c = \pi(path)$ **then**
            if $u_c = \psi(path)$ **then**
                if $u_c \in \lambda_0$ **then**
                    $x_c \leftarrow x_0$ // *Reinitializing the model*
                    $\Theta_c \leftarrow \{i \in \Omega : \tilde{p}_i[1] = u_c\}$ // *Reinitializing the model*
                    $e\_c \leftarrow 0$ // *Reinitializing the model*
                    Break the for loop // *The EBAM was reinitialized*
                else
                    Stop the algorithm, since a fault was detected
            else
                $\Theta_c \leftarrow \Theta_c \setminus \{path\}$ // *Removing a non-viable path*
    if $e\_c > 0$ **then** // *No break*
        if $\Theta_c = \emptyset$ **or** ($\|\Theta_c\| = 1$ **and** $e\_c < \omega$(the only path of Θ_c)) **then**
            Stop the algorithm, since a fault was detected
        $x_c \leftarrow \delta(x_c, \sigma)$ // *Updating the current state*
    $u_p \leftarrow u_c$ // *Updating the previous I/O status*

In order to elucidate the Algorithm 2, it will be explained line by line. The input is just the EBAM and the output is the fault detection. In Line 1, the system and the model are synchronized. The system must be in one of the initial states of the paths and the model must be in its initial state. In Line 2 is created the variable u_p that will store the previous value of the I/O status. In Line 3 is defined the actual state of the model x_c, that is, at the beginning, the initial one, then, x_c is x_0. In Line 4 the set of allowed path indexes of the whole play is defined and it will be updated along the play. In Line 5 is defined the counter e_c that will be used to counter the number of events after the initial state. In line 6, the *while* block, is to run the loop until a fault is detected without creating a variable just for it. The first step to detect faults is read the I/O status and this is done in Line 7. The observation is done until a different I/O status (comparing to u_c) appear. After it, in Line 8, the event that caused this transition is retrieved. And, since a event was obtained, the event counter e_c is updated in Line 9. In Line 10, the *if* block is the condition **C1** being applied, *i.e.*, if the event executed in the system was not possible in the model, a fault is detected and the algorithm ends. After this check, in Line 12, the allowed indexes path is updated, since associated to each transition, there is a set of allowed indexes path.

In Line 13, the *for* block will test every path in Θ_c. In Line 14, the *if* block verifies the conditions **C4** and **C5**. Thus, it starts verifying if e_c is equals to the number of events in the path. If it is not, nothing is done since no information was obtained. However, if it is, then, the path goes to the next verification, in Line 15. It checks if the path reached its last vertex. If not, this path is not valid anymore and it is removed from the allowed index path set in Line 24. If all the paths are removed, this means that the condition **C4** was violated and the fault will be detected at the end of this *for* block. However, it the path reached its last vertex, then, it proceeds to the final verification, in Line 16. The *if* block checks the conditions **C5** to evaluate if the model is reinitializable by checking if the last I/O status is an initial I/O status for any path. If it is not, then, a fault is detected and the algorithms ends. Otherwise, the model is reinitialized in Lines 17 to 20, *i.e.*, the current state becomes the initial one, the allowed index paths is evaluated now considering u_c as the initial I/O status and the event counter is set to 0.

After the *for* block, in Line 25, the *if* block verifies if the model was reinitialized by checking the event counter. If it was, e_c should be at least 1. In this case, in Line 26, the conditions **C2** and **C3** are verified. If at least one of them failed, then, a fault is detected and the algorithm ends. Otherwise, in Line 28, the current state x_c is updated, since the event is viable. Finally, in Line 29, the previous I/O status is updated as the current I/O status.

To illustrate the use of the Algorithm 2, an example is given in the sequel.

Example 3.4 (EBAM Fault Detection)
Considering the EBAM for $k = 1$ of the Example 3.3, let us evaluate, using Algorithm 2, if a fault has occurred according to the following observed paths:

$$\begin{cases} po_1 = ((3,4),\ (3),\ (),\ (1),\ ()) \\ po_2 = ((),\ (1),\ (),\ (3),\ (4),\ (3)) \\ po_3 = ((3),\ (1,3),\ (3),\ (2,3),\ (2)) \end{cases}$$

The algorithm analyze each event of each path given, so, at the first I/O status of the first path, there is no event, since a event is defined as the difference between two consecutive I/O status. Therefore, at the first I/O status of the first path, the algorithm make some definitions to be able to identify the fault as the current I/O status, the current state (in this case is the initial one), the path estimation (the set formed by the possible indexes paths, once it is possible to have more than one initial state) and the event counter (starts at 0).

Now, analyzing the nexts I/O status is possible to get the events. To clarify the fault detection, three paths will be analyzed in the sequel.

1) Analyzing the second I/O status of po_1:

At this point: $u_p = (3,4)$, $x_c = x_0$, $\Theta_c = \{1\}$, $e\_c = 0$

After waiting the observation (Line 7): $u_c = (3)$

Evaluating the event (Line 8): $\sigma = f_c^{-1}((3) - (3,4)) = \downarrow 4 = \sigma_1$

Updating the event counter (Line 9): $e\_c = e\_c + 1 = 0 + 1 = 1$

Checking if the event is active in the actual state (Line 10): $\sigma_1 \in \gamma(x_c = x_0) \Rightarrow$ *No fault*

Updating the path estimation (Line 12): $\Theta_c = \Theta_c \cap \theta(x_c = x_0, \sigma = \sigma_1) = \{1\} \cap \{1\} = \{1\}$

Checking the paths of $\|\Theta_c\|$ *(Line 14):* $e\_c = 1 \neq \pi(1) = 4 \Rightarrow$ *Finish the checking*

Verifying some conditions (Line 26): $e\_c > 0 \wedge \|\Theta_c\| = 1 \wedge e\_c = 1 > \omega(1) = 0 \Rightarrow$ *No fault*

Updating the current state (Line 28): $x_c = \delta(x_c, \sigma) = \delta(x_0, \sigma_1) = x_1$

Updating the previous I/O status (Line 29): $u_p = u_c = (3)$

After the analysis of the second I/O status, no fault was detected and the algorithm will do the same analysis for the others I/O status. After the last event of po_1, no fault was detected and po_1 is a fault-free path that reinitialize the model.

2) Analyzing the fourth I/O status of po_2:

At this point: $u_p = (), x_c = x_4, \Theta_c = \{2, 3\}, e\_c = 2$

After waiting the observation (Line 7): $u_c = (3)$

Evaluating the event (Line 8): $\sigma = f_c^{-1}((3) - ()) = \uparrow 3 = \sigma_6$

Updating the event counter (Line 9): $e\_c = e\_c + 1 = 2 + 1 = 3$

Checking if the event is active in the actual state (Line 10): $\sigma_6 \notin \gamma(x_4) \Rightarrow$ *Fault*

A fault was detected since in x_4 the only active event is σ_5 and the actual event is σ_6. Therefore, po_2 is a faulty path.

3) Analyzing the last I/O status of po_3:

At this point: $u_p = (2, 3), x_c = x_5, \Theta_c = \{2, 3\}, e\_c = 3$

After waiting the observation (Line 7): $u_c = (2)$

Evaluating the event (Line 8): $\sigma = f_c^{-1}((2) - (2, 3)) = \downarrow 3 = \sigma_2$

Updating the event counter (Line 9): $e\_c = e\_c + 1 = 3 + 1 = 4$

Checking if the event is active in the actual state (Line 10): $\sigma_2 \in \gamma(x_5) \Rightarrow$ *No fault*

Updating the path estimation (Line 12): $\Theta_c = \Theta_c \cap \theta(x_c = x_5, \sigma = \sigma_2) = \{2, 3\} \cap \{3\} = \{3\}$

Checking the paths of $\|\Theta_c\|$ (Line 14): $e\_c = 4 == \pi(1) = 4 \Rightarrow$ *Do the next verification*

Checking final I/O status (Line 15): $u_c = (2) == \psi(3) = (2) \Rightarrow$ *Do the next verification*

Checking if the model is reinitializable (Line 16): $u_c = (2) \notin \lambda_0 = \{(3, 4), (), (3)\} \Rightarrow$ *Fault*

A fault was detected since the last I/O status is not the initial I/O status for any path, therefore, the model is not reinitializable. Thus, po_3 is a faulty path since it does not allow the model to be reinitialized.

Chapter 4

Practical Example

4.1 System Description

The system used to implement the EBAM theory is a distribution system and it is shown in Figure 4.1. Its main goal is to sort the boxes (short and tall ones) that arrive in the distribution module and, then, distribute them to the correct conveyor: tall boxes to the left conveyor and short boxes to the right conveyor.

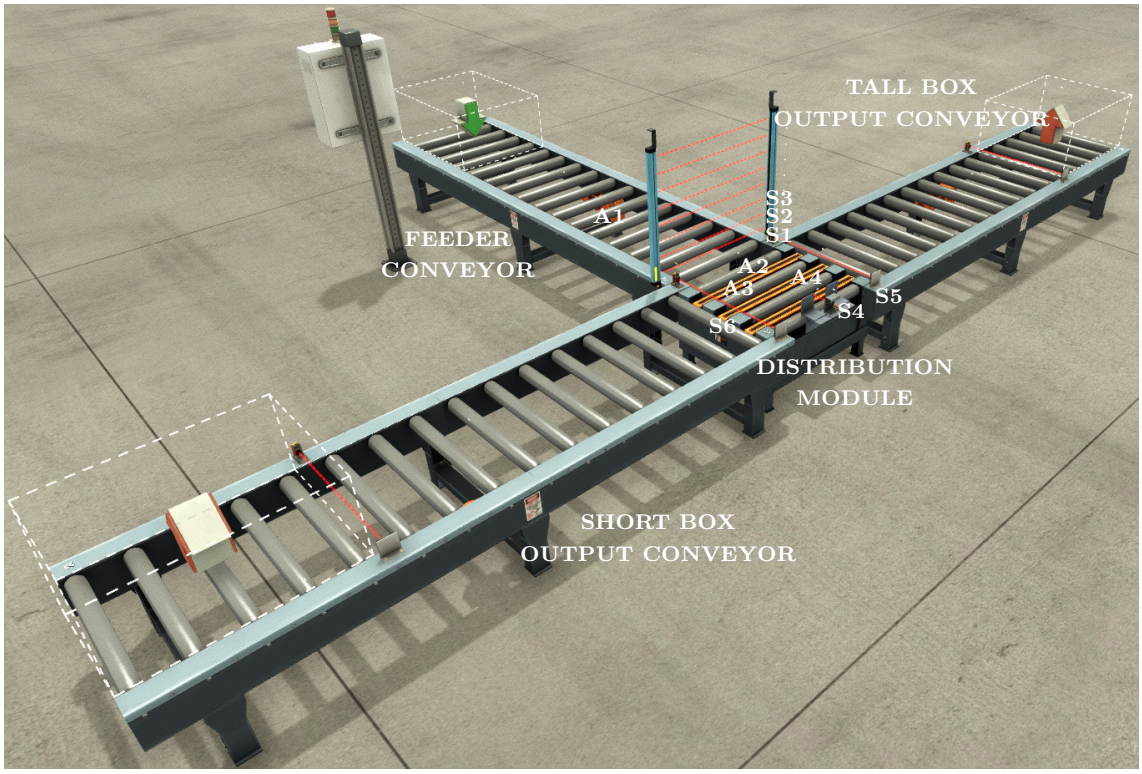

Figure 4.1: Sorting unit system.

The system was implemented using the 3D simulation software Factory I/O and controlled by a real PLC (Siemens S7-1200). The system is composed of 6 sensors and 4 actuators as described as follows in Table 4.1.

Table 4.1: System description.

ID	Type	Name	Description
$S1$	P Sensor	Pallet Sensor	Detects a pallet in the entry conveyor
$S2$	P Sensor	Low Sensor	Detects short and tall boxes
$S3$	P Sensor	High Sensor	Detects tall boxes
$S4$	P Sensor	Loaded	Detects if the box reached the end of the distribution conveyor
$S5$	N Sensor	At Left Entry	Detects when the box leaves the distribution conveyor to the left
$S6$	N Sensor	At Right Entry	Detects when the box leaves the distribution conveyor to the right
$A1$	Actuator	Conveyor Entry	The entry conveyor
$A2$	Actuator	Load	Moves forward the distribution conveyor
$A3$	Actuator	Transf. Left	Moves the distribution conveyor to the left
$A4$	Actuator	Transf. Right	Moves the distribution conveyor to the right

The fault-free behavior of the system is described as follows:

- The initial state there is no boxes on the conveyors and all actuators are turned off (0);
- The system starts after pressing the start button;
- The entry conveyor and the box feeder (which has a random time between 2 up to 5 seconds to feed the system with a random, short or tall, box) are turned on (1);
- When a box reaches the net sensors ($S1$, $S2$ and $S3$), it is detected if the box is short or tall;
- The box proceeds to the distribution conveyor that is turned on (moving forward) and the entry conveyor is turned off;
- When the box reaches the loaded sensor ($S4$) the distribution conveyor is turned off (stop moving forward);

- If the box is short, the distribution conveyor is turned on to the left and the left conveyor is turned on (if it is turned off), otherwise, if it is a tall box, then, the conveyor is turned on to the right and the right conveyor is turned on (if it is turned off);
- As soon as the box leaves the distribution conveyor, the entry conveyor is turned on again.

4.2 Modelling

In order to identify the model of the system using the EBAM, a crucial step is the observation. The system was implemented in Factory I/O as a virtual system and the control was carried out by the Siemens PLC S7-1200, that can be seen in Figure 4.2.

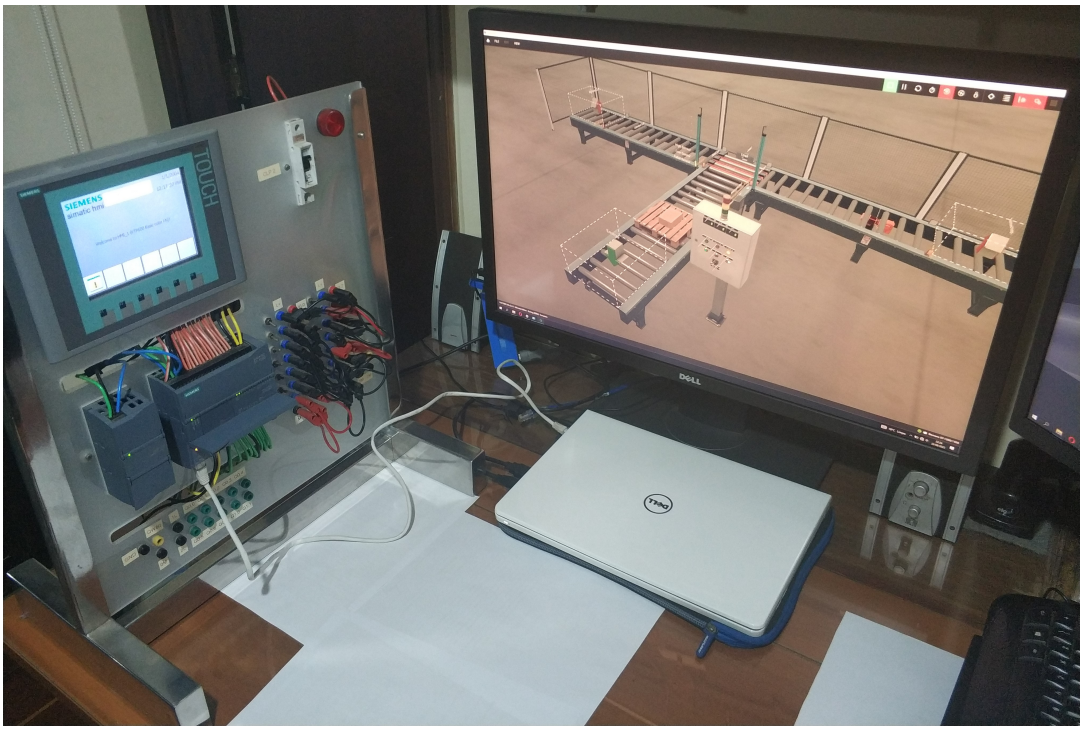

Figure 4.2: The Siemens PLC S7-1200 used for the practical example.

With the aim of identifying the system reading the data from sensors and actuators, a Python program was developed to read (and write) data from the PLC using the S7 protocol.

The computer used to observe the system, generate the model and simulate the faults was a notebook Dell Inspiron 5458, 1 TB HDD (5400 RPM), 8 GB RAM (DDR3L 1333 MHz), Intel Core i5-5200 (2.20 GHz) and Windows 10 (build 1903) OS.

To generate the model, three steps must be followed:

1) Observe the system and acquire the data;
To observe the system, it is necessary to create an observation vector, that is, an I/O vector, u. In this case, $u = [S1, S2, S3, S4, S5, S6, A1, A2, A3, A4]$. The total time of the acquisition was 10 days and it was saved as text files. The criterium used was the

2) Convert data to the EBAM format;
The inputs of the EBAM construction algorithm are the EBAM paths (Definition 3.7). Each EBAM path is a vector whose first element is the initial I/O status and the others elements are modified events of a given path. As a reminder, a path is a task or a cycle of the system. Therefore, before converting the data, it is necessary to define what are tasks or cycles in this case. It was defined that a task starts by the rising edge of the conveyor entry actuator ($\uparrow 7$) and finishes by the rising edge of the left or right entry ($\uparrow 4$ or $\uparrow 5$). To convert the acquisition to EBAM paths, another Python program was made and the total elapsed time was 119 s for 1282 EBAM paths. It is important to mention that is necessary to verify for each new path created if it already exists, since each path must be unique. It is important to mention that from the whole acquisition, if the paths were not verified, the total number of paths would be 38284.

3) Follow the EBAM construction algorithm.
Another Python program was made to generate the model following the Algorithm 1. The model was generated for $k = 1, 2$ and their information can be seen in Table 4.2.

Table 4.2: Model information.

k	Paths	States	Events	Transitions	Elapsed Time (ms)	
					Model	Distinguish Paths
1	1282	74	73	265	335	1133
2	1282	266	73	458	374	1142

Since the models ($k = 1, 2$) are huge, it will be shown here just a part of the model for $k = 1$. To avoid the several transitions in the model, the model will be generated only for the first 9 paths of the system. Their EBAM paths $\tilde{p}_1$, $\tilde{p}_2$, ..., $\tilde{p}_9$ are as follows:

$$\begin{cases}
\tilde{p}_1 = ((5,6),\ \uparrow 7,\ \uparrow 1,\ \uparrow 8,\ \uparrow 2,\ \downarrow 2,\ \downarrow 1,\ \uparrow 4 \uparrow 9 \downarrow 7 \downarrow 8,\ \downarrow 5,\ \downarrow 4,\ \uparrow 5) \\
\tilde{p}_2 = ((5,6,9),\ \uparrow 7 \downarrow 9,\ \uparrow 1,\ \uparrow 8,\ \uparrow 2,\ \downarrow 2,\ \downarrow 1,\ \uparrow 4 \uparrow 9 \downarrow 7 \downarrow 8,\ \downarrow 5,\ \downarrow 4,\ \uparrow 5) \\
\tilde{p}_3 = ((5,6,9),\ \uparrow 7 \downarrow 9,\ \uparrow 1,\ \uparrow 8,\ \uparrow 2 \uparrow 3,\ \downarrow 2 \downarrow 3,\ \downarrow 1,\ \uparrow 4 \uparrow 10 \downarrow 7 \downarrow 8,\ \downarrow 6,\ \downarrow 4,\ \uparrow 6) \\
\tilde{p}_4 = ((5,6,10),\ \uparrow 7 \downarrow 10,\ \uparrow 1,\ \uparrow 8,\ \uparrow 2 \uparrow 3,\ \downarrow 2 \downarrow 3,\ \downarrow 1,\ \uparrow 4 \uparrow 10 \downarrow 7 \downarrow 8,\ \downarrow 6,\ \downarrow 4,\ \uparrow 6) \\
\tilde{p}_5 = ((5,6,10),\ \uparrow 7 \downarrow 10,\ \uparrow 1 \uparrow 8,\ \uparrow 2,\ \downarrow 2,\ \downarrow 1,\ \uparrow 4 \uparrow 9 \downarrow 7 \downarrow 8,\ \downarrow 5,\ \downarrow 4,\ \uparrow 5 \uparrow 7 \downarrow 9) \\
\tilde{p}_6 = ((5,6,7),\ \uparrow 1 \uparrow 8,\ \uparrow 2,\ \downarrow 2,\ \downarrow 1,\ \uparrow 4 \uparrow 9 \downarrow 7 \downarrow 8,\ \downarrow 5,\ \downarrow 4,\ \uparrow 5 \uparrow 7 \downarrow 9) \\
\tilde{p}_7 = ((5,6,9),\ \uparrow 1,\ \uparrow 8,\ \uparrow 2 \uparrow 3,\ \downarrow 2 \downarrow 3,\ \downarrow 1,\ \uparrow 4 \uparrow 10 \downarrow 7 \downarrow 8,\ \downarrow 6,\ \downarrow 4,\ \uparrow 6) \\
\tilde{p}_8 = ((5,6,10),\ \uparrow 7 \downarrow 10,\ \uparrow 1 \uparrow 8,\ \uparrow 2,\ \downarrow 2,\ \downarrow 1,\ \uparrow 4 \uparrow 9 \downarrow 7 \downarrow 8,\ \downarrow 5,\ \downarrow 4,\ \uparrow 5) \\
\tilde{p}_9 = ((5,6,10),\ \uparrow 7 \downarrow 10,\ \uparrow 1,\ \uparrow 8,\ \uparrow 2,\ \downarrow 2,\ \downarrow 1,\ \uparrow 4 \uparrow 9 \downarrow 7 \downarrow 8,\ \downarrow 5,\ \downarrow 4,\ \uparrow 5 \uparrow 7 \downarrow 9)
\end{cases}$$

To simplify visualization, the following definitions are made:

$\sigma_{01} = \uparrow 7$, $\sigma_{02} = \uparrow 1$, $\sigma_{03} = \uparrow 8$, $\sigma_{04} = \uparrow 2$, $\sigma_{05} = \downarrow 2$, $\sigma_{06} = \downarrow 1$, $\sigma_{07} = \uparrow 4 \uparrow 9 \downarrow 7 \downarrow 8$, $\sigma_{08} = \downarrow 5$, $\sigma_{09} = \downarrow 4$, $\sigma_{10} = \uparrow 5$, $\sigma_{11} = \uparrow 7 \downarrow 9$, $\sigma_{12} = \uparrow 2 \uparrow 3$, $\sigma_{13} = \downarrow 2 \downarrow 3$, $\sigma_{14} = \uparrow 4 \uparrow 10 \downarrow 7 \downarrow 8$, $\sigma_{15} = \downarrow 6$, $\sigma_{16} = \uparrow 6$, $\sigma_{17} = \uparrow 7 \downarrow 10$, $\sigma_{18} = \uparrow 1 \uparrow 8$, $\sigma_{19} = \uparrow 5 \uparrow 7 \downarrow 9$

$\iota_{01} = \{1\}$, $\iota_{02} = \{1, 2, 3, 4, 7, 9\}$, $\iota_{03} = \{1, 2, 9\}$, $\iota_{04} = \{1, 2, 5, 6, 8, 9\}$, $\iota_{05} = \{1, 2, 8\}$, $\iota_{06} = \{2, 3\}$, $\iota_{07} = \{3, 4, 7\}$, $\iota_{08} = \{4, 5, 8, 9\}$, $\iota_{09} = \{4, 9\}$, $\iota_{10} = \{5, 8\}$, $\iota_{11} = \{5, 6, 8\}$, $\iota_{12} = \{5, 6, 9\}$, $\iota_{13} = \{6\}$, $\iota_{14} = \{7\}$

Replacing the events values in $\tilde{p}_i$ for $i = 1, \ldots, 9$ the representations are now as follows:

$$\begin{cases}
\tilde{p}_1 = ((5,6),\ \sigma_{01},\ \sigma_{02},\ \sigma_{03},\ \sigma_{04},\ \sigma_{05},\ \sigma_{06},\ \sigma_{07},\ \sigma_{08},\ \sigma_{09},\ \sigma_{10}) \\
\tilde{p}_2 = ((5,6,9),\ \sigma_{11},\ \sigma_{02},\ \sigma_{03},\ \sigma_{04},\ \sigma_{05},\ \sigma_{06},\ \sigma_{07},\ \sigma_{08},\ \sigma_{09},\ \sigma_{10}) \\
\tilde{p}_3 = ((5,6,9),\ \sigma_{11},\ \sigma_{02},\ \sigma_{03},\ \sigma_{12},\ \sigma_{13},\ \sigma_{06},\ \sigma_{14},\ \sigma_{15},\ \sigma_{09},\ \sigma_{16}) \\
\tilde{p}_4 = ((5,6,10),\ \sigma_{17},\ \sigma_{02},\ \sigma_{03},\ \sigma_{12},\ \sigma_{13},\ \sigma_{06},\ \sigma_{14},\ \sigma_{15},\ \sigma_{09},\ \sigma_{16}) \\
\tilde{p}_5 = ((5,6,10),\ \sigma_{17},\ \sigma_{18},\ \sigma_{04},\ \sigma_{05},\ \sigma_{06},\ \sigma_{07},\ \sigma_{08},\ \sigma_{09},\ \sigma_{19}) \\
\tilde{p}_6 = ((5,6,7),\ \sigma_{18},\ \sigma_{04},\ \sigma_{05},\ \sigma_{06},\ \sigma_{07},\ \sigma_{08},\ \sigma_{09},\ \sigma_{19}) \\
\tilde{p}_7 = ((5,6,9),\ \sigma_{02},\ \sigma_{03},\ \sigma_{12},\ \sigma_{13},\ \sigma_{06},\ \sigma_{14},\ \sigma_{15},\ \sigma_{09},\ \sigma_{16}) \\
\tilde{p}_8 = ((5,6,10),\ \sigma_{17},\ \sigma_{18},\ \sigma_{04},\ \sigma_{05},\ \sigma_{06},\ \sigma_{07},\ \sigma_{08},\ \sigma_{09},\ \sigma_{19}) \\
\tilde{p}_9 = ((5,6,10),\ \sigma_{17},\ \sigma_{02},\ \sigma_{03},\ \sigma_{04},\ \sigma_{05},\ \sigma_{06},\ \sigma_{07},\ \sigma_{08},\ \sigma_{09},\ \sigma_{19})
\end{cases}$$

Using Algorithm 1 for these 9 EBAM paths, the model generated (20 states, 19 events, 26 transitions and 2 ms to be constructed) is shown in Figure 4.3.

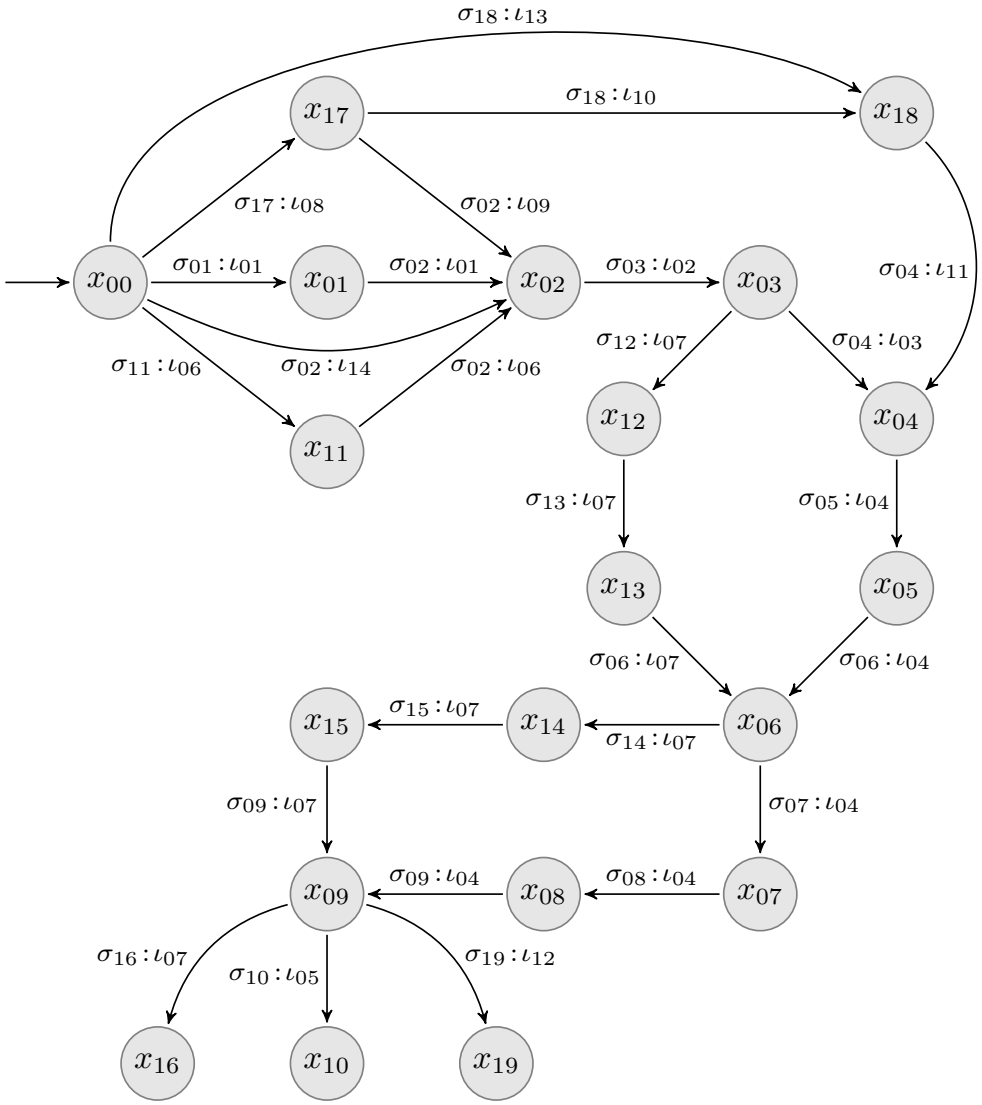

Figure 4.3: EBAM for $k = 1$ of the first 9 paths of the practical example.

To show that the paths are in the model, the second path will be played in the model. The transitions of the model that represents the transitions of the second path are highlighted in red and dashed in Figure 4.4.

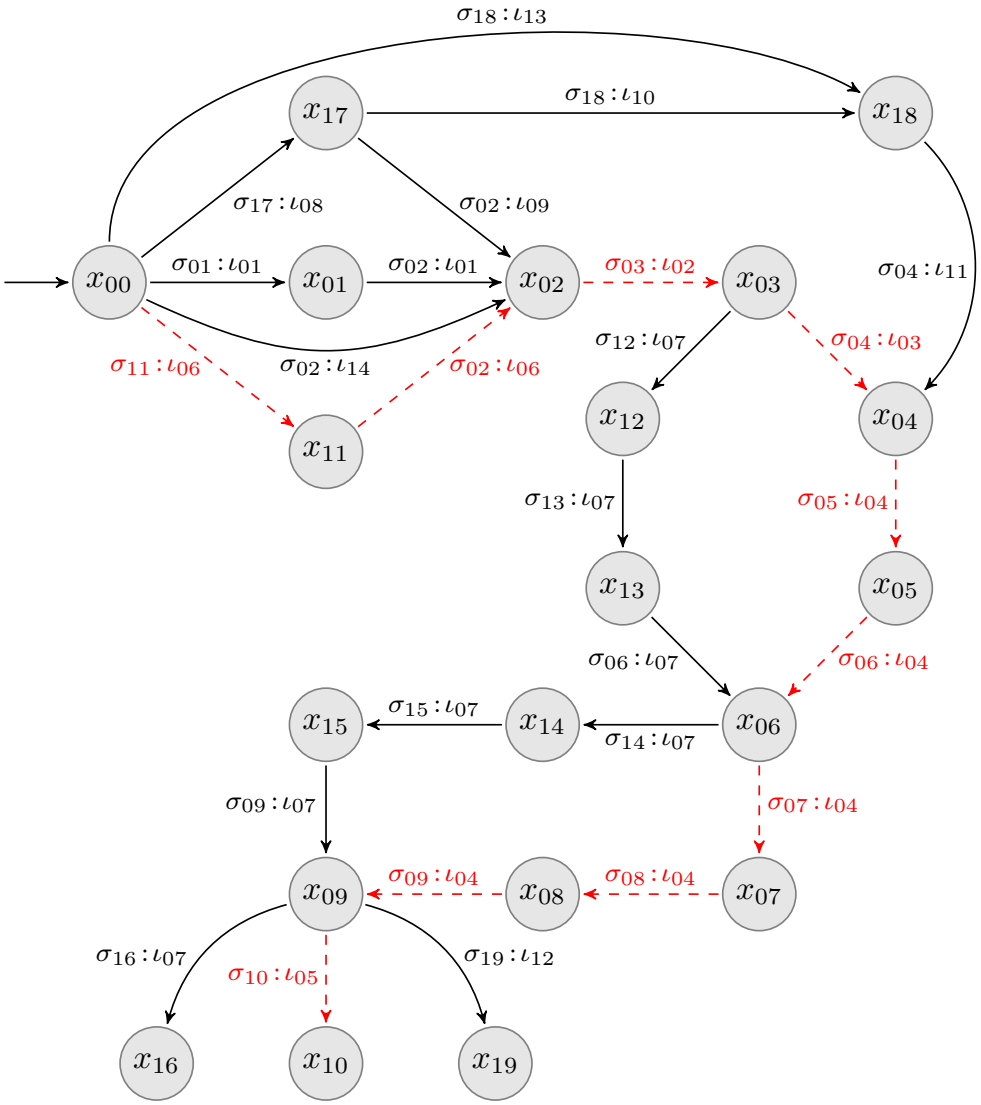

Figure 4.4: EBAM for $k = 1$ of the first 9 paths of the practical example highlighting the transitions of the second path.

Once all the transitions are, in fact, in the model, it is necessary now to check if the path during these transitions is allowed. That is, it is necessary to check every indexes set associated to these transitions. The index of the second path, 2, must be in all of these sets. The indexes sets are ι_i for $i = 2, 3, \ldots, 6$ and since 2 is in all of them, the play is valid and the second path is, indeed, represented in the model.

4.3 Fault Detection

A Python program was created to do the fault detection online following the Algorithm 2. The PLC was connected to the Factory I/O to control the system and connected to the Python program to send the I/O status. So, there were two connections at the same time and this might cause some delays in the I/O status acquistion that might miss some event, but this was verified and did not happen.

In order to analyze the efficiency of the fault detection using EBAM, 44 faults scenarios were designed to be tested in the practical example. A fault scenario simulates permanent or intermittent faults in sensors (one or more), actuators (one or more) or both. All the scenarios can be seen in Table 4.3.

Table 4.3: Fault scenarios of the practical example.

Scenario	Description	Detected		Scenario	Description	Detected	
		$k=1$	$k=2$			$k=1$	$k=2$
1	$S1_0$	Y	Y	23	$S3_{0,1,0}$	Y	Y
2	$S1_1$	Y	Y	24	$S4_0$	N_D	N_D
3	$S1_1$	Y	Y	25	$S4_1$	Y	Y
4	$S1_{0,1,0}$	Y	Y	26	$S4_1$	Y	Y
5	$S1_{0,1,0}$	Y	Y	27	$S4_{0,1,0}$	Y	Y
6	$S1_0, S2_0$	Y	Y	28	$S4_{0,1,0}$	Y	Y
7	$S1_1, S2_1$	Y	Y	29	$S5_1$	Y	Y
8	$S1_0, S3_0$	Y	Y	30	$S5_{1,0,1}$	Y	Y
9	$S1_1, S3_1$	Y	Y	31	$S5_{1,0,1}$	Y	Y
10	$S1_0, S2_0, S3_0$	Y	Y	32	$S6_1$	Y	Y
11	$S1_1, S2_1, S3_1$	Y	Y	33	$S6_{1,0,1}$	Y	Y
12	$S2_0$	Y	Y	34	$S6_{1,0,1}$	Y	Y
13	$S2_1$	Y	Y	35	$A1_0$	N_D	N_D
14	$S2_1$	Y	Y	36	$A1_1$	Y	Y
15	$S2_{0,1,0}$	Y	Y	37	$A1_1, S4_0$	Y	Y
16	$S2_{0,1,0}$	Y	Y	38	$A2_0$	Y	Y
17	$S2_0, S3_0$	Y	Y	39	$A2_1$	Y	Y
18	$S2_1, S3_1$	Y	Y	40	$A2_0, S4_0$	Y	Y
19	$S3_0$	N_L	N_L	41	$A3_0$	N_D	N_D
20	$S3_1$	Y	Y	42	$A3_1$	Y	Y
21	$S3_1$	Y	Y	43	$A4_0$	N_D	N_D
22	$S3_{0,1,0}$	Y	Y	44	$A4_1$	Y	Y

The notation adopted in the "Description" column in Table 4.3 describes the sensors and actuators used in the fault scenario and the fault type. Let E be a signal and f a fault associated to E. The fault can be divided in four types: (i) ON failure (permanent); (ii) OFF failure (permanent); (iii) ON, OFF, ON failure (intermittent) and (iv) OFF, ON, OFF failure (intermittent). Thus, E_1, E_0, $E_{1,0,1}$ and $E_{0,1,0}$ are the representation of the fault type (i), (ii), (iii) and (iv), respectively. Each element is separeted by a comma. For example, the description of the scenario 18, $S2_1, S3_1$, means that the fault occurred in the sensors S2 and S3 and both are ON failure. In addition, when a scenario has failure in several signals, the faults are not necessarily synchronous in time and when some scenarios have the same signals to simulate a fault, they were simulated at different states of the system.

The notation adopted in the "Detected" column in Table 4.3 describes if the fault was detected or not. Y means the fault was detected, $\mathrm{N_L}$ means the fault was not detected and the system was in a state with feasible events and, finally, $\mathrm{N_D}$ means the fault was not detected and the system was in a deadlock state.

Table 4.3 shows that both models were identically in terms of fault detection. In all 44 scenarios, they were able to detect 39 fault occurrences. The 5 non-detected faults can be explained. First, in scenario 19, the High Sensor was always 0, therefore, when a tall box arrives, it was not able to detect and the system interpreted that the box was a short one, thus, it went to the wrong conveyor. The other 4 faults leads the system to deadlocks, then, there is no way that these faults would be detected.

In Table 4.4, all the results are summarized and the efficiency is computed. It shows that the faults were detected with a 88.63 % efficiency for $k = 1, 2$.

Table 4.4: Fault detection results of the practical example.

	$k = 1$	$k = 2$
Detected faults	39	39
Non-detected faults	5	5
Efficiency (%)	88.63	88.63

Since there is no way to detect a non-detectable fault, the efficiency is now computed considering only the detectable faults and the results can be seen in Table 4.5.

Table 4.5: Fault detection results of the practical example considering only the detectable faults.

	$k = 1$	$k = 2$
Detected faults	39	39
Non-detected faults	0	0
Efficiency (%)	100	100

In this practical example, considering only the detectable faults, EBAM was able to detect all of them. Note that it is possible to detect some non-detectable faults if EBAM uses timing information. For example, if the system is at a deadlock state, it is possible to detect this fault, since an event is expected to occur within a certain time interval and, in this case, no event will occur and the timing information would detect a fault.

Chapter 5

Conclusions

In this work, a new model for DES black-box identification with the aim of fault detection is proposed. The main difference from this model, called Event-Based Automaton Model (EBAM), to the others proposed in the literature is that the EBAM is ruled by events instead of states. This difference implies in an advantage that allows the paths to have different initial I/O status, which is impossible to be modeled using the the other models proposed in the literature (all paths must have the same initial and final I/O status).

The EBAM is defined and the identification algorithm is presented. As the other models, EBAM uses paths observed from the fault-free behavior of the system to construct the model. Since EBAM is based on events, it is necessary to modify them to be EBAM paths, *i.e.*, paths where the first element is the initial I/O status of the paths (the first element of the original paths) and the other elements are event changes (consecutive difference between the I/O status of the paths). Then, with the EBAM paths, the identification algorithm can be used and the model is constructed. To illustrate how the identification works, some examples constructing the model were given.

In the sequel, it is shown that EBAM holds some important properties: (i) it uses a free parameter k; (ii) it uses a path estimation function; (iii) the observed language is a subset of the identified language, *i.e.*, the EBAM simulates the observed fault-free system behavior, which is crucial for any identified model meant to be used to detect faults; (iv) it is k-complete; (v) the exceeding language is the empty set if the model is acyclic. In addition, for a given value of k, the EBAM is, in general, more compact than the other models, which may increase the exceeding language, but this increase is not significant in some cases.

It is important to mention that all the paths were observed without adding any extra code to the PLC. It was done by creating a Python program that communicates

to the PLC using the Profinet protocol. Thus, the data was read from the PLC using an external program, *i.e.*, not using the manufacturer software (Totally Integrated Automation). The core idea was to simulate a real situation where it is necessary to observe the data from real PLCs in industry. The only code in the PLC was the LADDER code to control the system and there is no need to know it.

After proving the EBAM properties, a fault detection scheme and a fault detection algorithm were proposed. As in the DAOCT, the EBAM has conditions for an event be viable in the fault detection. For both models, the conditions are all analogous, except for the reinitialization condition, that is new. Since EBAM reinitializes after the end of a path, it is necessary to verify if it is possible. In DAOCT, the first and the final vertex of the paths are equal, so, there is no need to do this verification. To show the algorithm application and some fault detection ideas behind the EBAM, examples were provided.

Finally, all the theory presented was implemented in a practical example, where the system is a virtual one implemented in Factory I/O and the control is carried out by a Siemens PLC S7-1200. The system was automatized, the LADDER was implemented and, then, the observation was carried out. Even for days of observation, the model was computed quickly and the fault detection was a successful with 88.63 % efficiency for the 44 fault scenarios tested if considering all faults and 100 % efficieny considering only the detectable faults. Using other identified models were not possible in this case, since the paths have different initial I/O status. It is important to mention that all the faults not detected are, in fact, not detectable, in this practical example.

The EBAM fills an important gap in industry, because it is usual that the processes have different initial and final I/O status for the paths and all the procedure presented is possible to be implemented in industry. The model is computed quickly and the fault detection is done online. In case of false alarms, the path that raised it can be added to the model (no need to recompute the entire model) and this false alarm will not be raised again.

Possible directions for future research could be: (*i*) add time information to EBAM to allow it to detect faults that lead the system to deadlocks (and also other fault types); (*ii*) define residue functions to be able to isolate the faults; (*iii*) use artificial intelligence to allow the model to classify a new behavior observed as a fault-free or a faulty one; (*iv*) extend the model to accept continuous variables such as temperature and pressure.

References

[1] SAMPATH, M., SENGUPTA, R., LAFORTUNE, S., et al. "Diagnosability of discrete-event systems", *IEEE Transactions on automatic control*, v. 40, n. 9, pp. 1555–1575, 1995.

[2] DEBOUK, R., LAFORTUNE, S., TENEKETZIS, D. "Coordinated decentralized protocols for failure diagnosis of discrete event systems", *Discrete Event Dynamic Systems*, v. 10, n. 1-2, pp. 33–86, 2000.

[3] QIU, W., KUMAR, R. "Decentralized failure diagnosis of discrete event systems", *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, v. 36, n. 2, pp. 384–395, 2006.

[4] MOREIRA, M. V., JESUS, T. C., BASILIO, J. C. "Polynomial time verification of decentralized diagnosability of discrete event systems", *IEEE Transactions on Automatic Control*, v. 56, n. 7, pp. 1679–1684, 2011.

[5] ZAYTOON, J., LAFORTUNE, S. "Overview of fault diagnosis methods for discrete event systems", *Annual Reviews in Control*, v. 37, n. 2, pp. 308–320, 2013.

[6] SANTORO, L. P., MOREIRA, M. V., BASILIO, J. C. "Computation of minimal diagnosis bases of Discrete-Event Systems using verifiers", *Automatica*, v. 77, pp. 93–102, 2017.

[7] RAN, N., GIUA, A., SEATZU, C. "Enforcement of diagnosability in labeled Petri nets via optimal sensor selection", *IEEE Transactions on Automatic Control*, v. 64, n. 7, pp. 2997–3004, 2018.

[8] CABRAL, F. G., MOREIRA, M. V. "Synchronous Diagnosis of Discrete-Event Systems", *IEEE Transactions on Automation Science and Engineering*, v. 17, n. 2, pp. 921–932, 2019.

[9] VIANA, G. S., MOREIRA, M. V., BASILIO, J. C. "Codiagnosability analysis of discrete-event systems modeled by weighted automata", *IEEE Transactions on Automatic Control*, v. 64, n. 10, pp. 4361–4368, 2019.

[10] VIANA, G. S., BASILIO, J. C. "Codiagnosability of discrete event systems revisited: A new necessary and sufficient condition and its applications", *Automatica*, v. 101, pp. 354–364, 2019.

[11] HU, Y., MA, Z., LI, Z. "Design of supervisors for active diagnosis in discrete event systems", *IEEE Transactions on Automatic Control*, 2020.

[12] MACHADO, T. H. D. M. C., VIANA, G. S., MOREIRA, M. V. "Event-based automaton model for identification of discrete-event systems for fault detection", *Control Engineering Practice*, 2022. Manuscript submitted for publication.

[13] EL MEDHI, S. O., LECLERCQ, E., LEFEBVRE, D. "Petri nets design and identification for the diagnosis of discrete event systems". In: *2006 IAR Annual Meeting*. Citeseer, 2006.

[14] MOREIRA, M. V., LESAGE, J.-J. "Fault diagnosis based on identified discrete-event models", *Control Engineering Practice*, v. 91, pp. 104101, 2019.

[15] CABASINO, M. P., GIUA, A., SEATZU, C. "Identification of Petri nets from knowledge of their language", *Discrete Event Dynamic Systems*, v. 17, n. 4, pp. 447–474, 2007.

[16] RAMÍREZ-TREVIÑO, A., RUIZ-BELTRÁN, E., RIVERA-RANGEL, I., et al. "Online fault diagnosis of discrete event systems. A Petri net-based approach", *IEEE Transactions on Automation Science and Engineering*, v. 4, n. 1, pp. 31–39, 2007.

[17] DOTOLI, M., FANTI, M. P., MANGINI, A. M. "Real time identification of discrete event systems using Petri nets", *Automatica*, v. 44, n. 5, pp. 1209–1219, 2008.

[18] ESTRADA-VARGAS, A. P., LOPEZ-MELLADO, E., LESAGE, J.-J. "A comparative analysis of recent identification approaches for discrete-event systems", *Mathematical Problems in Engineering*, v. 2010, 2010.

[19] DOTOLI, M., FANTI, M. P., MANGINI, A. M., et al. "Identification of the unobservable behaviour of industrial automation systems by Petri nets", *Control Engineering Practice*, v. 19, n. 9, pp. 958–966, 2011.

[20] ESTRADA-VARGAS, A. P., LÓPEZ-MELLADO, E., LESAGE, J.-J. "A black-box identification method for automated discrete-event systems", *IEEE Transactions on Automation Science and Engineering*, v. 14, n. 3, pp. 1321–1336, 2015.

[21] SAIVES, J., FARAUT, G., LESAGE, J.-J. "Automated partitioning of concurrent discrete-event systems for distributed behavioral identification", *IEEE Transactions on Automation Science and Engineering*, v. 15, n. 2, pp. 832–841, 2017.

[22] CABASINO, M. P., GIUA, A., HADJICOSTIS, C. N., et al. "Fault model identification and synthesis in Petri nets", *Discrete Event Dynamic Systems*, v. 25, n. 3, pp. 419–440, 2015.

[23] KLEIN, S., LITZ, L., LESAGE, J.-J. "Fault detection of discrete event systems using an identification approach", *IFAC Proceedings Volumes*, v. 38, n. 1, pp. 92–97, 2005.

[24] ROTH, M., LESAGE, J.-J., LITZ, L. "An FDI method for manufacturing systems based on an identified model", *IFAC Proceedings Volumes*, v. 42, n. 4, pp. 1406–1411, 2009.

[25] MOREIRA, M. V., LESAGE, J.-J. "Discrete event system identification with the aim of fault detection", *Discrete Event Dynamic Systems*, v. 29, n. 2, pp. 191–209, 2019.

[26] MOOR, T., RAISCH, J., O'YOUNG, S. "Supervisory control of hybrid systems via l-complete approximations", *Proc. WODES98*, pp. 426–431, 1998.

[27] CASSANDRAS, C. G., LAFORTUNE, S. *Introduction to discrete event systems*. Springer Science & Business Media, 2009.

[28] CURY, J. E. R. "Teoria de controle supervisório de sistemas a eventos discretos", *V Simpósio Brasileiro de Automação Inteligente (Minicurso)*, 2001.

[29] DE SOUZA, R. P., MOREIRA, M. V., LESAGE, J.-J. "Fault detection of Discrete-Event Systems based on an identified timed model", *Control Engineering Practice*, v. 105, pp. 104638, 2020.

Appendix A

Codes

The main codes used in this master's thesis were developed to overcome practical problems that arose during the research. There are four main problems:

1) Data acquisition;
   Read data from the PLC without add any extra code to it.

2) Split the data into paths;
   Since EBAM uses paths, it is necessary to split the whole acquisition into paths (or it will be a single path, that it is not the ideia). In addition, the paths used in EBAM are unique, therefore, it is important to not create a path if it already exists.

3) Compute the EBAM;
   Compute the EBAM using the paths following the Algorithm 1.

4) Online fault detection.
   Detect a fault while the system is running following the Algorithm 2.

All the codes were developed in Python 3.8 with the ideia to build a library and not single scripts. There is also a software with Graphical User Interface (GUI), but EBAM it not implemented since EBAM was created after this software. All these codes can be seen in `https://github.com/thiagohmcm/Masters-Thesis.git`.

At the beginning of each class and function there is an explanation about its objective and input and output variables. It is important to mention that the whole code, *i.e.*, variable names, filenames and comments are in Brazilian Portuguese, however, if needed, an english version can be done in the future.

A.1 Data acquisition

To do the data acquisition it is necessary, first, to create a connection to the PLC to be able to read data directly from it. Therefore, it was developed a library that handle connections to some Siemens PLCs (S7-300, S7-400, S7-1200 and S7-1500) using the S7 protocol based on the library Snap7 (http://snap7.sourceforge.net/). The code that handle the connection is called *clp_siemens_s7.py*.

A.2 Split the data into paths

After the acquisition is done, it is necessary to split the paths, since EBAM use them. The strategy adopted is the usage of hashs. Each path has a hash associated to it and if two paths share the same hash, then, they are equal. At each path generated the hash is compared to the others hashs and if it already exists, then, this path is discarded and the process goes on untill the end of the acquisition file. The code that create hashs is called *obter_hashs.py*, the one that compare the paths is called *comparar_caminhos.py* and, finally, the code that split the data into paths is called *separar_caminhos.py*.

A.3 Compute the EBAM

The code that compute the EBAM using the paths is called *modelo_ebam.py*.

A.4 Online fault detection

The code that run the online fault detection is called *deteccao_de_falhas_online.py*.

A.5 Software with GUI

This software, called *CLPlant*, was developed at the beginning of the research with the purpose of: (*i*) help the study of some systems in Factory I/O and (*ii*) implement something practical. The idea is to maintain this software implementing other models and features. This software is available in Brazilian Portuguese and in English, however, to change the language it is necessary to change the main code (the ideia is to implement a button in the future). It is important to highlight that the data acquisition was done using this software. The main code to run the software is called *clplanta.py*.