



ENFORCING CURRENT-STATE OPACITY THROUGH SHUFFLE IN EVENT OBSERVATIONS

Raphael Julio Barcelos

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: João Carlos dos Santos Basilio

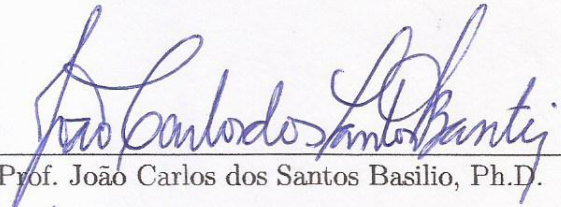
Rio de Janeiro
Março de 2018

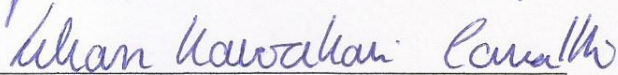
ENFORCING CURRENT-STATE OPACITY THROUGH SHUFFLE IN EVENT
OBSERVATIONS

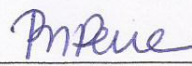
Raphael Julio Barcelos

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

Examinada por:


Prof. João Carlos dos Santos Basilio, Ph.D.


Prof. Lilian Kawakami Carvalho, D.Sc.


Prof. Patrícia Nascimento Pena, D.Eng.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2018

Barcelos, Raphael Julio

Enforcing current-state opacity through shuffle in event observations/Raphael Julio Barcelos. – Rio de Janeiro: UFRJ/COPPE, 2018.

XI, 61 p.: il.; 29, 7cm.

Orientador: João Carlos dos Santos Basilio

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2018.

Referências Bibliográficas: p. 58 – 61.

1. Discrete event systems. 2. Opacity. 3. Opacity-enforcement. 4. Event observation delay. I. Basilio, João Carlos dos Santos. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Acknowledgments

I would like to thank God for guiding and enlightening me throughout my life.

I thank my parents, Sheyla and Milton, for their love, comprehension, support and also for always prioritizing my education.

I thank my long time friends, Manuela Sena and David Goes, for having encouraged me to start and finish my M.Sc. degree and for all pleasant days we spent together.

I thank Jéssica Vieira, Waldez Júnior and Pâmela Lopes for sharing with me their joy, lunch time and spare time whenever it was possible.

I specially thank Gustavo Viana, Marcos Vinicius and Ingrid Antunes, for the discussions and suggestions regarding our researches.

I thank my advisor João Carlos Basilio for your dedication to teach me.

I thank the friends of Laboratory of Control and Automation (LCA), specially, Prof. Lilian Kawakami Carvalho, Felipe Cabral, Antonio Gonzalez, Alexandre Gomes, Juliano Freire, Públio Lima and Wesley Silveira.

I thank all of my friends from Brasília, specially, Alexandre Meuren, Marcos Nihari and Bruna Castelo.

I thank Hudson Marcondes, Kadmo Keijock and Igor Cohen, for the time we shared our home in Paradiso.

I thank the National Council for Scientific and Technological Development (CNPq) for the financial support.

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

IMPOSIÇÃO DE OPACIDADE DE ESTADO ATUAL POR MEIO DE TROCAS DAS OBSERVAÇÕES DE EVENTOS

Raphael Julio Barcelos

Março/2018

Orientador: João Carlos dos Santos Basilio

Programa: Engenharia Elétrica

Opacidade é uma propriedade que garante que qualquer comportamento secreto do sistema permaneça escondido de um Intruso. Neste trabalho será considerado o problema da opacidade de estado atual e será proposto um Forçador de Opacidade capaz de permutar adequadamente a ordem de observação dos eventos ocorridos no sistema, de tal forma que o Intruso seja enganado e sempre estime, erroneamente, pelo menos um estado não secreto. Condições necessárias e suficientes para a síntese do Forçador de Opacidade são propostas a fim de que a mesma seja factível e são também apresentados dois algoritmos para construção do autômato que implementa a estratégia usada pelo Forçador de Opacidade.

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

ENFORCING CURRENT-STATE OPACITY THROUGH SHUFFLE IN EVENT OBSERVATIONS

Raphael Julio Barcelos

March/2018

Advisor: João Carlos dos Santos Basilio

Department: Electrical Engineering

Opacity is a property that ensures that a secret behavior of the system is kept hidden from an Intruder. In this work, we deal with current-state opacity, and propose an Opacity-Enforcer that is able to change, in an appropriate way, the order of observation in the event occurrences in the system, so as to mislead the Intruder to always wrongly estimate at least one non-secret state. A necessary and sufficient condition for the feasibility of the Opacity-Enforcer synthesis is presented and also two algorithms to build the automaton that realizes such an enforcement.

Contents

List of Figures	viii
Lista de Símbolos	ix
1 Introduction	1
2 Background	6
2.1 Discrete Event Systems	6
2.1.1 Language	7
2.1.2 Automaton	9
2.2 Opacity	15
3 Opacity-Enforcement methodology	23
3.1 Problem Formulation	24
3.2 The Opacity-Enforcement Strategy	25
3.3 Algorithms	33
3.3.1 Computation of automaton D	34
3.3.2 An Algorithm for Opacity Enforcement Checking	39
3.3.3 An Algorithm for Finding Minimal Delay Bounds	45
3.4 Example	48
4 Conclusion and future works	56
Bibliography	58

List of Figures

2.1	Graph of an automaton.	10
2.2	Automata used in Examples 2.5, 2.6 and 2.7.	13
2.3	Accessible part of automaton G_2 , $Ac(G_2)$	13
2.4	Parallel composition $G = G_1 G_2$	13
2.5	Observer automaton $G_{obs} = O_{bs}(G_3, \Sigma_o)$	14
2.6	System related to the examples 2.10, 2.13, 2.14 and 2.15.	18
2.7	System related to the examples 2.11 and 2.12.	18
2.8	Observer of automaton depicted in Figure 2.6.	20
3.1	The Opacity-Enforcement architecture.	23
3.2	Automata used in Examples 3.2.	28
3.3	Automaton D	38
3.4	Mouse in a Maze problem structure.	49
3.5	Automaton G that models the system dynamics.	49
3.6	Observer automaton $G_{obs} = O_{bs}(G, \Sigma_o)$	49
3.7	Part of V that shows undesirable states.	51
3.8	Part of V that shows decision conflicts.	52
3.9	Automaton R_{oe} that realizes the opacity enforcement strategy.	52
3.10	Estimator Automaton \mathcal{E}	54
3.11	Automaton G	55
3.12	Opacity-Enforcer automaton of G	55

Lista de Símbolos

$Ac(G)$	Accessible part of an automaton G , p. 10
D	Automaton which models changes in the order of event observation, p. 36
G_{int}	Automaton which models the Intruder's state estimates, p. 39
G_{obs}	Observer automaton $O_{bs}(G, \Sigma_o)$, p. 12
G_{sys}	Automaton which models the observable event occurrence in the system, p. 39
$L(G)$	Language generated by automaton G , p. 9
L/s	Post-language of L after a sequence s , p. 8
L_e	Extended language generated by the action of the Opacity Enforcer, p. 29
L_{obs}	Language generated by the observer automaton G_{obs} , p. 21
$OE(s_e)$	Opacity-Enforcement function over the extended sequence s_e , p. 28
$O_{bs}(G, \Sigma_o)$	Observer automaton of G with respect to Σ_o , p. 11
$P_{a,b}(s)$	Projection of sequence s of Σ_a over Σ_b , p. 7
$P_{a,b}^{-1}(s)$	Inverse projection of sequence s , p. 7

$Pre(s)$	Prefix-closure of a sequence s , p. 7
$R(\sigma)$	Rename operation over event σ , p. 28
R_{oe}	Opacity-Enforcer automaton, p. 38
SD	Step delay bounds, p. 30
S_p	Sequence permutation of s , p. 26
$T(s)$	Tuple mapping of a sequence s , p. 25
$T^{-1}(t)$	Inverse tuple mapping of a tuple t , p. 25
$UR(x, \Sigma_o)$	Unobservable reach of state x with respect to Σ_o , p. 11
V	Verifier automaton used to check CSO enforcement, p. 39
V_{obs}	Observer automaton of V with respect to Σ_o , p. 39
X_s	Set of secret states with respect to automaton G , p. 16
$X_{s,obs}$	Set of secret states with respect to automaton G_{obs} , p. 21
Σ^*	Kleene-Closure of event set Σ , p. 7
Σ^n	Tuple of order n whose elements are events, p. 25
Σ_e	Extended set of events, p. 28
$\Sigma_{o,s}$	Set of event observations with respect to Σ_o , p. 28
Σ_o	Set of observable events, p. 11
Σ_{uo}	Set of unobservable events, p. 11
\mathbb{Z}_+^*	Positive Integers, p. 34
\mathbb{Z}_+	Non negative integers, p. 6
\mathcal{E}	Estimator Automaton, p. 53

$\mathcal{P}(t)$	Permutation mapping of a tuple t , p. 25
ε	Empty sequence, p. 7
$cut(q)$	Elimination function, p. 34
$rep(q, i)$	Replacement function, p. 34

Chapter 1

Introduction

In recent years, opacity has emerged in the Discrete Event Systems (DES) community [1] as a convenient way to deal with certain security issues. It is a property that ensures that a secret behavior of the system is kept hidden from external observers, usually referred to as Intruders. Different definitions of opacity and the characterization of all of these notions have been proposed in the literature, most of them recalled in an overview over opacity by JACOB *et al.* [2], which contains also some reviews on verifications and on extensions of opacity to probabilistic models.

It is worth remarking that there exist different notions of opacity, depending on how the secret behavior of the system is defined, the most usual ones being as follows: strong/weak opacity [3], language-based opacity (LBO) [4, 5], current-state opacity (CSO) [1, 6], initial-state opacity (ISO) [1, 7], initial-and-final-state opacity (IFO) [8], K -Step Opacity [6], and ∞ -Step Opacity [9].

Several works have addressed different problems related to opacity. Opacity verification has been considered in [10],[11], [12], [13], [14], [9] and [15], where, given a system and a secret behavior, its main concern is to check whether opacity holds or not. These works differ in respect to which opacity notion is being considered, what strategy is being used to verify opacity, and the computational cost to achieve the proposed verification. Another opacity problem addressed is the so called opacity enforcement, which has been considered by YIN and LAFORTUNE [16], WU and LAFORTUNE [17, 18, 19], FALCONE and MARCHAND [20, 21], DUBREIL

et al. [5], SABOORI and HADJICOSTIS [22], CASSEZ *et al.* [23], TONG *et al.* [24]. Supervisory control theory was used in DUBREIL *et al.* [5], SABOORI and HADJICOSTIS [22], YIN and LAFORTUNE [16], and TONG *et al.* [24] in order to restrict the system behavior to the non-secret behavior only, being, therefore, a conservative approach. In a different context, opacity is enforced by manipulating the observable behavior [17–19, 23]. To this end, CASSEZ *et al.* [23] propose an opacity enforcement strategy by means of static and dynamic mask, where, in the static case, a maximum subset of the observable events that makes the system opaque is computed, and, in the dynamic case, a dynamic mask is synthesized with a view to turning on and off the events observations, depending on the previously observed behavior and also on the secrets.

Another opacity enforcement strategy, through insertion function, was developed in WU and LAFORTUNE [17], and enhanced in WU and LAFORTUNE [18] for the case where the intruder discovers the insertion function, and, optimized in WU and LAFORTUNE [19] with respect to the developed algorithms. This strategy consists of inserting fictitious event signals in the output of the system so as to make the secret behavior look like the non-secret behavior from the Intruder’s perspective.

With respect to other notions of opacity, FALCONE and MARCHAND [20, 21] proposed a strategy to enforce K -Step Opacity, where, depending on the already observed behavior, it would hold the incoming event signals for an exact amount of steps such that the system becomes opaque.

As far as opacity enforcement is concerned, an strategy which is performed through manipulations of the observable behavior, the so-called edit function was proposed in [25]. The edit function is an extension of the insertion functions, in the sense that it also allows events to be erased, being proposed in the context of enforcing privacy while preserving utility. Privacy is a more general concept that embeds opacity and utility means that a given behavior must not have any other observationally equivalent behavior, from the point of view of external observers. In contrast to the strategies that use supervisory control theory, where the system

behavior is constrained, when manipulating the observable behavior, the system is allowed to run freely, while the Intruder is misled by the observation changes.

Two recent works also deals with opacity enforcement [26, 27]. Following along the line of insertion functions, JI and LAFORTUNE [26] explore the edit function and propose an enforcement strategy such that intruders cannot infer the secrets even though they know about the used enforcement policies. In addition, constraints are jointly used with edit functions in order to avoid trivial solutions. However, opacity enforceability may not hold depending on the chosen constraints. In contrast with the previous works concerning insertion functions, KEROGLOU and LAFORTUNE [27] propose a new insertion function which is embedded into the system and acts based on the real location of the system, being a more powerful strategy than the previously related ones.

Other problems related to this opacity were addressed, for example, in SABOORI and HADJICOSTIS [11], where the sensor selection problem is considered in order to ensure opacity. Another important results were presented by BEN-KALEFA and LIN [28], where opacity closure under union and intersection was explored, and, hence, the problems of finding opaque sublanguages and superlanguages were also addressed.

The extension of opacity notions to timed DES was presented by CASSEZ [10], where it has been shown that opacity verification is undecidable, even for a restrictive class of Timed Automata.

Among the most usual notions of opacity we chose to work here with CSO (current-state opacity), *i.e.*, for every secret state that can be reached from an initial state trough some sequence, there must exist a non-secret state reached from the same or another initial state through some sequence, having these two sequences the same observation from the Intruder's point of view. We propose a new opacity enforcement strategy that is based on changes in the order of observations of the events occurred in the system, so as to mislead the Intruder to always wrongly estimate at least one non-secret state.

The proposed opacity enforcement strategy works as follows. When the Opacity-Enforcer receives a signal associated with an event occurrence, it may release immediately this signal or hold it until one or more events occur. In other words, the Opacity-Enforcer is responsible for, after receiving an event, to choose, based on preset rules, either to release or hold it until the arrival of other events, with a view to changing the order of the observation of the events. The events released by the Opacity-Enforcer are transmitted either through a second set of channels or a wifi connection susceptible to leak information to Intruders. The idea behind the opacity enforcement strategy proposed here is to shuffle the event observations in the same way as that modeled by the automaton, first presented by NUNES *et al.* [29], to address delays in communication channels between the system and a diagnoser to synchronize all possible changes in the order of observations, in order for the Intruder to always believe that it is in at least one non-secret state by means of its estimates.

This work differs from [17, 20, 23, 25] in the following sense: *(i)* since we only delay the observation of certain events in order to mislead the Intruder's estimations, our work differs from [20] in the sense that, in [20], all event releases are held in the imminence of leaking a secret and, once the system has entered in the non-secret behavior, all of the held information is released; *(ii)* as far as [23] is concerned, we do not choose a minimal set of observable events as static masks and we cannot turn off the observation of any observable event as the dynamical masks proposed in [23] can; we do not insert fictitious events, as in [17], but shuffle the order of event observations; and, *(iv)* being able to insert and delete event signals without any constraints except for the existence of the utility behavior, as performed in [25], give us enough degree of freedom to simulate any behavior we want, thus, the lack of the utility behavior makes the edit function a powerful but not realistic tool.

This work is organized as follows. Chapter 2 presents a brief review on DES theory and the notation used throughout the work, as well as a short overview on opacity containing its most usual notions. Chapter 3 embeds the problem for-

mulation, the opacity-enforcement strategy and the algorithms developed here to synthesize the Opacity-Enforcer, and, hence, enforce CSO over the system. Finally, Chapter 4 summarizes all of the contributions of this work and suggests possible extensions.

Chapter 2

Background

This chapter is intended to ensure a better comprehension of the results presented in this work. The main definitions of DES will be explained, some basic concepts concerning finite-state automaton (FSA) used throughout the text will be recalled and an overview of opacity will be presented. This chapter is structured as follows. In Section 2.1 we present a brief review on discrete event systems theory focusing mainly on the topics necessary for the comprehension of the work developed here. In Section 2.2, we present and explain the main notions of opacity, defining formally each one of these notions.

2.1 Discrete Event Systems

In contrast to the Continuous Variable Systems (CVS) which are characterized by continuous-states and time-driven transitions, Discrete Event Systems are characterized by having its state space as a discrete set and the transitions between the states are event-driven [30].

Definition 2.1 (Discrete Event System) *A Discrete Event System (DES) is a discrete-state, event-driven system, that is, its state evolution depends entirely on the occurrence of asynchronous discrete events over time.*

Example 2.1 shows the difference between modeling a system in CVS and in

DES:

Example 2.1 *Assume that the system under consideration is a water tank being the water level the variable we want to analyze. Besides the tank, the system is also composed of a pump to fill it and a drain to empty it. The state is the water level, which is clearly a continuous variable, since the water level goes up and down according to the control laws; note also that the change in the water level takes place over the time, thereafter a time-driven transition. Thus the system behavior must be modeled as a CVS. Now, let the system be a warehouse where we want to get track of the its number of packages in the store. Notice that this numbers increases as packages arrives and decreases as they are delivered. Thus, the state space of the variable, that represents the number of stored packages in the warehouse, is a discrete variable, and in addition the change of state is dictated by the arrival or delivery of packages, bearing no time dependence, i.e., state transitions are now event-driven, which implies that the warehouse must be modeled as a DES.*

2.1.1 Language

One way to formally model the behavior of DES is by using “languages”. To understand the concept of languages, we must first define the “set of events” $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_n\}$, $n \in \mathbb{Z}_+$, where $\mathbb{Z}_+ := \{z \in \mathbb{Z} : z \geq 0\}$ and each element of Σ is associated with an event occurrence in the system, and then define a “sequence” $s = \sigma_1\sigma_2 \dots \sigma_n$ (also called “word”, “string” or “trace” in the literature) as a concatenation of events and the “length of a sequence”, denoted as $|s| = n$, as the number of events in the sequence. A sequence with no events is denoted by ε and has zero length. The formal definition of a language is as follows:

Definition 2.2 *A language L defined over an event set Σ is a set of finite-length sequences formed from events $\sigma_i \in \Sigma$, $i = 1, 2, \dots, n$.*

In order to build sequences, we “concatenate” events with events or sequences with events, as it is explained in the following example.

Example 2.2 Let $\Sigma = \{a, b, c\}$, and assume that a system has its language L defined by all possible sequences with length 3 neither starting nor finishing with event “c”. Then we know that the sequences of L can start and finish with either “a” or “b”, and the second event occurrence can be any event from Σ . Thus we can first create all sequences of L with length 2 by concatenating events “a” and “b” with all events in Σ , generating “aa”, “ab”, “ac”, “ba”, “bb” and “bc”. Then we generate the sequences of length 3 by concatenating the previous sequences with events “a” and “b”, obtaining $L = \{aaa, aab, aba, abb, aca, acb, baa, bab, bba, bbb, bca, bcb\}$.

Remark 2.1 The empty sequence ε is the identity element of the concatenation operation, i.e., $s\varepsilon = \varepsilon s = s$.

The *Kleene-Closure* of an event set Σ is denoted by Σ^* and represents all possible sequences with finite length formed with the events of Σ and it includes ε . For example, if we have $\Sigma = \{a, b\}$, then $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$. Note that Σ^* is infinitely countable.

Before moving on to the operations that can be performed on sequences, we must first clarify some terms. Let $s = tuv$, where $s, t, u, v \in \Sigma^*$, we call t a prefix of s , u a substring of s and v a suffix of s . We now present some useful operations on sequences used throughout the text:

- *Prefix-closure* of a sequence s , defined as $Pre(s) := \{u \in \Sigma^* : (\exists v \in \Sigma^*)[uv = s]\}$.
- *Natural projection*, or simply projection, of a larger event set Σ_a over a smaller event set Σ_b , i.e., $\Sigma_b \subseteq \Sigma_a$, is defined as $P_{a,b} : \Sigma_a^* \rightarrow \Sigma_b^*$, where $P_{a,b}(\varepsilon) = \varepsilon$, $P_{a,b}(\sigma) = \sigma$ if $\sigma \in \Sigma_b$, $P_{a,b}(\sigma) = \varepsilon$ if $\sigma \in \Sigma_a \setminus \Sigma_b$ and $P_{a,b}(s\sigma) = P_{a,b}(s)P_{a,b}(\sigma)$, where $s \in \Sigma_a^*$, $\sigma \in \Sigma_a$.
- *Inverse projection*, defined as $P_{a,b}^{-1} : \Sigma_b^* \rightarrow \Sigma_a^*$, where $P_{a,b}^{-1}(t) := \{s \in \Sigma_a^* : P_{a,b}(s) = t\}$ and $\Sigma_b \subseteq \Sigma_a$.

The *prefix-closure*, *projection* and *inverse projection* can be extended over a language L by applying them to each sequence of L . Another useful operation on languages used throughout the text is the *post-language*.

- *Post-language* of L after a sequence s is defined as $L/s := \{t \in \Sigma^* : st \in L\}$.

Notice that if $s \notin L$, then $L/s = \emptyset$.

The following shows these operations on languages.

Example 2.3 Let $L_1 = \{abc, abb, acb, c\}$, $\Sigma_1 = \{a, b, c\}$, $\Sigma_2 = \{a, b\}$. We have then $Pre(L_1) = \{\varepsilon, a, c, ab, ac, abc, abb, acb\}$, $P_{1,2}(L_1) = \{\varepsilon, ab, abb\}$ and $L_1/a = \{bc, bb, cb\}$. Now, assume that $L_2 = \{\varepsilon, ab, abb\}$, then $P_{1,2}^{-1}(L) = \{c^*, c^*ac^*bc^*, c^*ac^*bc^*bc^*\}$

2.1.2 Automaton

Different formalisms can be used in the modeling of DES, such as Automaton, Petri Nets, Transition Systems, Timed Automaton, etc. Among these formalisms, only automata and Petri nets have the capability to model DES whose dynamics are expressed by languages. Although Petri net formalism is a tool more powerful than automaton (*e.g.*, some classes of infinite languages cannot be modeled by finite state automata but by Petri nets), automaton formalism is more intuitive and easier to understand. It is important to remark that infinite-state automaton do exists, although it would require infinite memory for its graphical representation. In our context, the finite-state automata are enough for our goals. Formally, automata can be defined as:

Definition 2.3 (Automaton) A automaton, denoted by G , is a six-tuple $G := (X, \Sigma, f, \Gamma, x_0, X_m)$ where X is the finite set of states, Σ is the finite set of events, $f : X \times \Sigma \rightarrow X$ is the state transition function, $\Gamma : X \rightarrow 2^\Sigma$ is the active event set function, being defined as $\Gamma(x) := \{\sigma \in \Sigma : f(x, \sigma)!\}$, where $f(x, \sigma)!$ denotes that $f(x, \sigma)$ is defined, $x_0 \in X$ is the initial state and $X_m \subseteq X$ is the set of marked states.

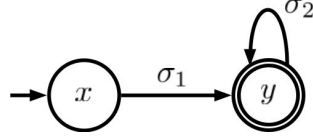


Figure 2.1: Graph of an automaton.

Such an automaton is usually referred to as deterministic automaton.

The transition function can be extended to $f : X \times \Sigma^* \rightarrow X$ by the following recursion: $f(x, s\sigma) = f(f(x, s), \sigma)$, $f(x, \epsilon) = x$, where $\sigma \in \Sigma$ and $s \in \Sigma^*$.

The graphical representation of an automaton, also known as state transition diagram, is composed by nodes, representing the states, and labeled arcs, representing the transitions between these states. Note also that the marked state is represented by double circles and the initial state is represented by the states with a single arrow pointing to it. Figure 2.1 shows a state transition diagram of automaton $G = (X, \Sigma, f, \Gamma, x_0, X_m)$, where $X = \{x, y\}$, $\Sigma = \{\sigma_1, \sigma_2\}$, $f(x, \sigma_1) = y$, $f(y, \sigma_2) = y$, $\Gamma(x) = \sigma_1$, $\Gamma(y) = \sigma_2$, $x_0 = x$ and $X_m = \{y\}$.

When modeling a system, we use the state marking in order to highlight states with special meaning, be it because these states are important in some way, or because they represent the completion of a desired task for example. When these pieces of information are not necessary, marked states can be omitted, which is the case of the present work. Therefore, we will explicitly omit X_m from the automaton tuples throughout the text.

As we mentioned before, languages and automata are strictly related, being the latter a way to graphically represent the former, when it is regular. In this regard, we have that the extraction of the language generated by an automaton can be obtained by inspecting the automaton state transition diagram and looking for sequences generated by all possible paths starting from the initial state. The formal definition is as follows:

Definition 2.4 (Generated language) *The language generated by an automaton $G = (X, \Sigma, f, \Gamma, x_0, X_m)$ is defined as $\mathcal{L}(G) := \{s \in \Sigma^* : f(x_0, s)!\}$.*

Example 2.4 *Let automaton G be the one depicted in Figure 2.1. It is not difficult to check that the language generated by G is $L(G) = Pre(\sigma_1\sigma_2^*)$.*

In this work we will denote the language generated by an automaton $L(G)$ simply as L . We now present a few operations with automata, used throughout Chapter 3 and whose knowledge is required to understand the proposed opacity-enforcement strategy.

Definition 2.5 (Accessible part) *The accessible part operation of an automaton $G = (X, \Sigma, f, \Gamma, x_0)$ deletes all states not reachable by any sequence from the initial state and is defined by: $Ac(G) := (X_{Ac}, \Sigma, f_{Ac}, \Gamma_{Ac}, x_0)$, where $X_{Ac} := \{x \in X : (\exists s \in \Sigma^*)[f(x_0, s) = x]\}$ and $f_{Ac} : X_{Ac} \times \Sigma^* \rightarrow X_{Ac}$, $f_{Ac}(x, \sigma) = f(x, \sigma)$ if $f(x, \sigma) \in X_{Ac}$ and $f(x, \sigma)$ is not defined otherwise.*

Taking the accessible part of an automaton is the same as checking which states are reachable from the initial state and then deleting those states that are not reachable. In order to compute the accessible part of an automaton, we must start from the initial state x , put it in a FIFO list, and search for all next states that are reached from x_0 . We, then, put the states that were found in the list. The next step is to remove the initial state from the list and analyzing the next element of it, searching for all next states reached from it, and putting them on the list, if they do not belong to it yet. We carry out this process for all states in the list until it becomes empty. The states which have not ever been an element of the list are the states not reachable, and so, must be deleted.

Definition 2.6 (Parallel composition) *The parallel composition between automata creates a new automaton where the common behaviors of the previous automata are synchronized and their individual characteristics run freely, being formally defined by $G_1||G_2 := Ac(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f_{1||2}, \Gamma_{1||2}, x_{0,1} \times x_{0,2})$, where:*

$$f_{1||2}((x_1, x_2), \sigma) := \begin{cases} (f_1(x_1, \sigma), x_2), & \text{if } \sigma \in \Gamma_1(x_1) \setminus \Sigma_2 \\ (x_1, f_2(x_2, \sigma)), & \text{if } \sigma \in \Gamma_2(x_2) \setminus \Sigma_1 \\ (f_1(x_1, \sigma), f_2(x_2, \sigma)), & \text{if } \sigma \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

When modeling a system, events determine state changes and, in order to acknowledge that change, sensors are used in the system, being responsible for recording event occurrences and for sending signals to the data acquisition system. Note that the fact that an event occurrence makes the state change does not imply that it necessarily has a sensor attached to it. We call observable events those events whose occurrences are noticeable to an outside watcher and, to this end, have sensors to record their occurrences. On the other hand, unobservable events are invisible to an outside watcher and, thus, there are no sensors to record their occurrences. Thus, from this point onwards, we will assume that the event set Σ is partitioned as $\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$, where Σ_o and Σ_{uo} are the sets of observable and unobservable events, respectively.

The outside watcher is also named “*observer*”, being modeled by an “*observer automaton*”, which captures all the observable behavior of a system. In order to completely define the *observer* of G , here denoted as $O_{bs}(G, \Sigma_o)$, the definition of *unobservable reach* $UR(x, \Sigma_o)$ is required, *i.e.*, $UR(x, \Sigma_o) := \{x' \in X : (\exists s \in \Sigma_{uo}^*) [f(x, s) = x']\}$. The concept of unobservable reach can be extended to a set of states $Y \in 2^X$ as $UR(Y, \Sigma_o) := \bigcup_{x \in Y} UR(x, \Sigma_o)$. Now that the unobservable reach was defined, we present the *observer automaton* G_{obs} as follows [30]:

Definition 2.7 (Observer Automaton) *The observer of automaton G with respect to a set of observable events Σ_o is defined by $G_{obs} = O_{bs}(G, \Sigma_o) := (X_{obs}, \Sigma_o, f_{obs}, \Gamma_{obs}, x_{0,obs})$, where $X_{obs} \in 2^X$, $f_{obs}(x_{obs}, \sigma) := \bigcup_{x \in x_{obs} \wedge f(x, \sigma) \in \Sigma_o} UR(f(x, \sigma), \Sigma_o)$, $\Gamma_{obs}(x_{obs}) := \bigcup_{x \in x_{obs}} \Gamma(x) \cap \Sigma_o$, $x_{0,obs} := UR(x_0, \Sigma_o)$. Finally, $\mathcal{L}(G_{obs}) = P_o(\mathcal{L}(G_{obs}))$ will be denoted by L_{obs} , where $P_o : \Sigma^* \rightarrow \Sigma_o^*$.*

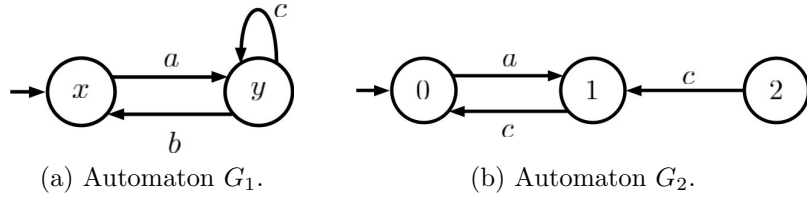


Figure 2.2: Automata used in Examples 2.5, 2.6 and 2.7.

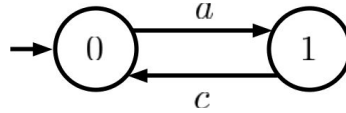


Figure 2.3: Accessible part of automaton G_2 , $Ac(G_2)$.

The next three examples show the afore mentioned operations applied to automata G_1 and G_2 , depicted in Figures 2.2a and 2.2b, respectively.

Example 2.5 *Let us compute $Ac(G_1)$. Notice that, in automaton G_1 , the initial state x reaches y and state y reaches x , that has already been checked. Therefore, all states of G_1 are reachable, and so, $Ac(G_1) = G_1$. Now, let us compute the accessible part of automaton G_2 . Firstly, we start from the initial state 0, which reaches state 1 only, and then, we check state 1, which has a unique transition to state 0 (already checked). Finally, since all states reached from the initial state 0 have been checked, we finish the operation. Note that state 2 has never been checked, since it was never reached from 0, thus, we must delete state 2 and all the transitions attached to it. The state transition diagram of automaton $Ac(G_2)$ is shown in Figure 2.3.*

Example 2.6 *Let us compute $G = G_1 || G_2$, where G_1 and G_2 are depicted in Figures 2.2a and 2.2b, respectively. Assume that the event sets of G_1 and G_2 are $\Sigma_1 =$*

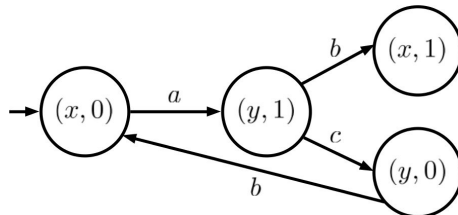


Figure 2.4: Parallel composition $G = G_1 || G_2$.

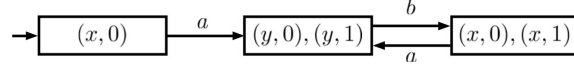


Figure 2.5: Observer automaton $G_{obs} = O_{bs}(G_3, \Sigma_o)$.

$\{a, b, c\}$ and $\Sigma_2 = \{a, c\}$. In order to compute automaton G , we first need to compare the sets of events to determine which events are “common” and which ones are “private”. Since $\Sigma_1 = \{a, b, c\}$ and $\Sigma_2 = \{a, c\}$, the “common events” between the automata are events “a” and “c”. Event “b” is a “private event” of G_2 . In order to ensure that the resulting automaton is already accessible, then we start building the parallel composition from the initial states of G_1 and G_2 , forming the G , which is given by $(x, 0)$. We now check the possible transitions from states x and 0 . Since there is only one transition in each automaton and both of them are labeled with the common event “a”, we have that state x evolves to y and state 0 evolves to 1 , i.e., state $(x, 0)$ evolves to $(y, 1)$ with event “a”. Again, in state $(y, 1)$, we check the possible transitions from states y and 1 . It can be seen that state y has a “self loop” with event “c” and that state 1 goes to 0 with event “c” too; then state $(y, 1)$ goes to $(y, 0)$ with event “c”. Still analyzing state $(y, 1)$, we have that state y evolves to x with event “b”, which is a private event, and so, state y goes to x while state 0 does not evolve. Thus, state $(y, 1)$ evolves to $(x, 1)$ with “b”. In state $(x, 1)$ we have that event “a” is in the active event set of x , i.e., $\Gamma_1(x) = \{a\}$, but it is not defined in the active event set of 1 , since $\Gamma_2(1) = \{c\}$. Since “a” and “c” are common events, they are not defined in state $(x, 1)$. Proceeding in this way for states $(y, 0)$ and $(x, 1)$, we obtain automaton G , whose state transition diagram is depicted in Figure 2.4.

Example 2.7 In this example, we want to build the observer automaton of G , depicted in Figure 2.4, with respect to the set of observable events $\Sigma_o = \{a, b\}$, which will be denoted as $G_{obs} = O_{bs}(G, \Sigma_o)$. First we take the initial state $x_0 = (x, 0)$ and compute its unobservable reach, which is $UR(x_0, \Sigma_o) = \{x_0\}$, and so, the initial state of the observer is $x_{0,obs} = \{(x, 0)\}$. After that, for all events $\sigma \in \Sigma_o$, we set $f_{obs}(\{(x, 0)\}, \sigma) = UR(f((x, 0), \sigma), \Sigma_o)$. Since only event “a” is defined in

$\{(x, 0)\}$, we have that $f_{obs}(\{(x, 0)\}, a) = UR(f((x, 0), a), \Sigma_o) = UR((y, 1), \Sigma_o) = \{(y, 0), (y, 1)\}$ since $(y, 1)$ is reached by event “a” and $(y, 0)$ is reached from $(y, 1)$ by the unobservable event “c”; thus, a new state $\{(y, 0), (y, 1)\}$ is created. We carry out the same procedure for state $\{(y, 0), (y, 1)\}$, from which, the only defined observable event is “b”, and so, $f_{obs}(\{(y, 0), (y, 1)\}, b) = UR(f((y, 0), b), \Sigma_o) \cup UR(f((y, 1), b), \Sigma_o) = UR((x, 0), \Sigma_o) \cup UR((x, 1), \Sigma_o) = \{(x, 0)\} \cup \{(x, 1)\} = \{(x, 0), (x, 1)\}$, thus state $\{(x, 0), (x, 1)\}$ is added to the observer and also the transition from $\{(y, 0), (y, 1)\}$ to it, labeled by “b”. Finally, we have that $f_{obs}(\{(x, 0), (x, 1)\}, a) = UR(f((x, 0), a), \Sigma_o) = UR((y, 1), \Sigma_o) = \{(y, 0), (y, 1)\}$, since $f((x, 1), a)$ is not defined. Note that state $\{(y, 0), (y, 1)\}$ already exists in the observer automaton, and so, we only add to the observer the transition from $\{(x, 0), (x, 1)\}$ to $\{(y, 0), (y, 1)\}$, labeled by event “a”. Figure 2.5 shows the resulting observer $G_{obs} = O_{bs}(G, \Sigma_o)$.

2.2 Opacity

Opacity is a general flow information property that characterizes whether a “secret”, with respect to some system behavior, can be discovered or not by an external observer, named Intruder. The notion of opacity can be split in weak opacity or strong opacity, depending on how much the Intruder is able to infer from the secret behavior of the system, as follows [3].

Definition 2.8 (Strong and Weak Opacities) *Given a system modeled by $G = (X, \Sigma, f, \Gamma, x_0)$, whose generated language is L , a general mapping $\Theta : \Sigma_a^* \rightarrow \Sigma_b^*$, where $\Sigma_a, \Sigma_b \subseteq \Sigma$, and two languages $L_1, L_2 \subseteq L$, then:*

- L_1 is strongly opaque with respect to L_2 and Θ if $\Theta(L_1) \subseteq \Theta(L_2)$;
- L_1 is weakly opaque with respect to L_2 and Θ if $\Theta(L_1) \cap \Theta(L_2) \neq \emptyset$.

Notice, from Definition 2.8, that strong opacity implies weak opacity, but not conversely. We illustrate the definitions of strong and weak opacities with the following example.

Example 2.8 Let $L = \text{Pre}(abc+cab)$ and consider the following sublanguages of L : $L_1 = \{abc\}$, $L_2 = \{a, ab, abc\}$, $L_3 = \{cab\}$. In addition, let $\Sigma_o = \{a, b\}$, $\Sigma_{uo} = \{c\}$, $\Sigma = \Sigma_o \cup \Sigma_{uo} = \{a, b, c\}$ and $\Theta = P_o : \Sigma^* \rightarrow \Sigma_o^*$. Notice that $P_o(L_1) = \{ab\}$, $P_o(L_2) = \{a, ab\}$, and $P_o(L_3) = \{ab\}$. Thus, we can say that L_1 is strongly opaque with respect to L_3 and $\Theta = P_o$, since $P_o(L_1) \subseteq P_o(L_3)$, and that L_2 is weakly opaque with respect to L_3 and $\Theta = P_o$, since $P_o(L_2) \cap P_o(L_3) = \{ab\} \neq \emptyset$. Note, however, that L_2 is not strongly opaque with respect to L_3 and $\Theta = P_o$, since $a \notin P_o(L_3)$.

Note that when strong and weak opacity definitions were proposed in LIN [3], its goal was to characterize opacity as a general property between languages and to show that other properties — anonymity, secrecy, observability, diagnosability and detectability — could be seen as special cases of opacity upon properly defining languages L_1 and L_2 ; that is the reason why neither secrets nor Intruder were mentioned in those definitions.

Opacity can also be characterized according to how we define what the secret behavior is, leading to six most usual types of opacity, one of them related to secret language and the others related to secret states, as follows [2, 8]: language-based opacity, current-state opacity, initial-state opacity, initial-and-final-state opacity, K -step opacity and ∞ -step opacity, where the latter five opacity notions will be referred to as state-based opacity.

We start by introducing the notion of language-based opacity, first presented by BADOUEL *et al.* [4], and formally defined by DUBREIL *et al.* [5]. In words, language-based opacity states that “given a secret language $L_s \subset L$ and a set of observable events Σ_o , then every secret sequence in the secret language $s \in L_s$ must have the same observation as some non-secret sequence $t \in L \setminus L_s$ ”. Formally, language-based opacity is defined as follows.

Definition 2.9 (Language-Based Opacity) Given a system modeled by $G = (X, \Sigma, f, \Gamma, x_0)$, projection $P_o : \Sigma^* \rightarrow \Sigma_o^*$, and a secret language $L_s \subseteq L$, where $L = \mathcal{L}(G)$, we say that G is language-based opaque with respect to Σ_o and L_s if $\forall s \in L_s$, there exists $t \in L \setminus L_s$ such that $P_o(s) = P_o(t)$, i.e., $P_o(L_s) \subseteq P_o(L \setminus L_s)$.

Notice that when comparing Definitions 2.8 and 2.9, it is easy to verify that language-based opacity is equivalent to strong opacity, if we set $L_1 = L_s$, $L_2 = L \setminus L_s$ and $\Theta = P_o$.

Example 2.9 *Let us consider here the same language $L = \text{Pre}(abc + cab)$ as in Example 2.8 and assume that $\Sigma_o = \{a, b\}$ and $\Sigma_{uo} = \{c\}$. Defining the secret language as $L_s = \{abc\}$, then, the system is language-based opaque, since $cab \in L \setminus L_s$ and $P_o(abc) = P_o(cab)$.*

We will now consider the notions of state-based opacity. We start with the notion of current state opacity (CSO), which was first introduced by BRYANS *et al.* [1] in the context of Petri Nets and was presented as a property of finite-state automata by SABOORI and HADJICOSTIS [6]. Stating that a system is current-state opaque is the same as being sure that “the intruder never knows if the system is actually in a secret state or not”. The formal definition is presented as follows.

Definition 2.10 (Current-State Opacity[8]) *Given a system modeled by $G = (X, \Sigma, f, \Gamma, x_0)$, projection $P_o : \Sigma^* \rightarrow \Sigma_o^*$, and a set of secret states $X_s \subseteq X$, G is current-state opaque with respect to Σ_o and X_s if for all $s \in L$ such that $f(x_0, s) \in X_s$, there exists $t \in L$ such that $f(x_0, t) \in X \setminus X_s$ and $P_o(s) = P_o(t)$.*

Example 2.10 *Let G be the automaton represented in Figure 2.6, $\Sigma = \{a, b, c\}$, $\Sigma_o = \{a, b\}$, $\Sigma_{uo} = \{c\}$ and $X_s = \{3\}$. We can say that G is current-state opaque because sequence “acb”, which reaches secret state 3, has the same projection as sequence “cab”, that reaches state 7, i.e., $f(0, acb) = 3 = X_s$, $f(0, cab) = 7 \in X \setminus X_s$ and $P_o(acb) = P_o(cab) = ab$. Note that, if we set $\Sigma_o = \{a, c\}$, $\Sigma_{uo} = \{b\}$ and $X_s = \{4\}$, the system is no longer current-state opaque, since there does not exist a sequence $t \in L : f(x_0, t) \in X \setminus X_s$, whose projection is $P_o(t) = P_o(acba) = aca$.*

The second notion of state-based opacity, the so called initial-state opacity, was also first defined in the context of Petri Nets by BRYANS *et al.* [1] and brought to the finite-state automaton formalism by SABOORI and HADJICOSTIS [7]. It

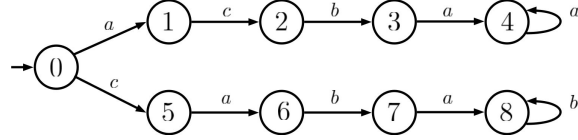


Figure 2.6: System related to the examples 2.10, 2.13, 2.14 and 2.15.

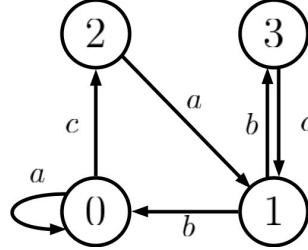


Figure 2.7: System related to the examples 2.11 and 2.12.

means that “the intruder is never sure whether the initial state of the system was a secret state or not”, being formally defined as follows.

Definition 2.11 (Initial-State Opacity) Given a system modeled by $G = (X, \Sigma, f, \Gamma, X_0)$, projection $P_o : \Sigma^* \rightarrow \Sigma_o^*$, and a set of secret initial states $X_{0,s} \subseteq X_0$, G is initial-state opaque with respect to Σ_o and $X_{0,s}$ if $\forall x_0 \in X_{0,s}$ and for all $s \in L$, $f(x_0, s)!$, there exists $y_0 \in X_0 \setminus X_{0,s}$ and $t \in L$, $f(y_0, t)!$ such that $P_o(s) = P_o(t)$.

Example 2.11 Let G be the automaton represented in Figure 2.7 with $\Sigma = \{a, b, c\}$, $\Sigma_o = \{a, b\}$, $\Sigma_{uo} = \{c\}$, $X_0 = X$ and $X_{0,s} = \{2\}$, which was presented in [7]. In this example, the intruder wants to discover if, among all the possible initial states, the system has started from a secret state. Indeed the system modeled by automaton G is initial-state opaque, since for every sequence s starting from state 2, there exists another sequence $t = cs$ starting from state 0 with the same observation, i.e., $P_o(s) = P_o(t)$. Now, assuming that the set of secret initial states is $X_{0,s} = \{0\}$, then the system is no longer initial-state opaque, since when the sequence “aa” is observed, the Intruder will track down the initial state 0.

The next notion of state-based opacity, called initial-and-final-state opacity (IFO), was presented by WU and LAFORTUNE [8] aiming to model anonymous communications, where the identities of both, the sender (initial state) and receiver (final

state), must be hidden from external observers (intruders). The system is said to be IFO if “the intruder, after receiving a sequence of observable events, is never sure whether the estimated pair of initial and final states is a secret pair or not”. IFO is formally defined as follows.

Definition 2.12 (Initial-and-Final-State Opacity) *Given a system modeled by $G = (X, \Sigma, f, \Gamma, X_0)$, projection $P_o : \Sigma^* \rightarrow \Sigma_o^*$, and a set of secret pair of states $X_{sp} \subseteq X_0 \times X$, G is initial-and-final-state opaque with respect to Σ_o and X_{sp} if $\forall (x_0, x_f) \in X_{sp}$ and for all $s \in L$, such that $f(x_0, s) = x_f$, there exists $(y_0, y_f) \in (X_0 \times X) \setminus X_{sp}$ and $t \in L$, $f(y_0, t) = y_f$, such that $P_o(s) = P_o(t)$.*

Example 2.12 *The same automaton G represented in Figure 2.7 with $\Sigma = \{a, b, c\}$, $\Sigma_o = \{a, b\}$ and $\Sigma_{uo} = \{c\}$ will be used to explain IFO. The system is initial-and-final-state opaque with respect to the secret pair of states $X_{sp} = \{(3, 1)\}$ but it is not initial-and-final-state opaque with respect to $X_{sp} = \{(0, 0)\}$. IFO holds for the former case because for every sequence $s \in L$, $f(3, s) = 1$, there is a sequence $t \in L : f(1, s) = 1$ with the same observation, since the transition from state 3 to state 1 is given by the unobservable event “c”, whereas in the latter case, where $X_{sp} = \{(0, 0)\}$, if “aa*” is observed, the Intruder will be sure that the system started from state 0 and is currently in it, since there is no other sequence t such that $P_o(t) = aa^*$ and $f(x_1, t) = x_2$, $(x_1, x_2) \neq (0, 0)$.*

Note that both CSO and ISO can be seen as a special case of IFO, if we set $X_{sp} = X_0 \times X_s$ for CSO and $X_{sp} = X_{0,s} \times X$ for ISO.

The next notion of state-based opacity described here is K -step opacity, proposed by SABOORI and HADJICOSTIS [6]. It is a more general property that embeds the secrecy of states from the last K steps executed by the system until the current moment, therefore it means that “the intruder is never sure that the system is in a secret state or has visited one in the last K steps”. K -step opacity is defined as follows.

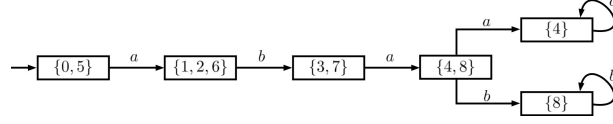


Figure 2.8: Observer of automaton depicted in Figure 2.6.

Definition 2.13 (K -Step Opacity) Given a system modeled by $G = (X, \Sigma, f, \Gamma, x_0)$, projection $P_o : \Sigma^* \rightarrow \Sigma_o^*$, a set of secret states $X_s \subseteq X$, and an integer $K \geq 0 \in \mathbb{Z}_+$, we say that G is K -step opaque with respect to Σ_o , X_s and K if for all $s \in L$, $f(x_0, s)!$ and for all $s' \in \text{Pre}(s)$ such that $|P(s) \setminus P(s')| \leq K$, $f(x_0, s') \in X_s$, there exists $t \in L$, $f(x_0, t)!$, and $t' \in \text{Pre}(t)$ such that $f(x_0, t') \in X \setminus X_s$, $P_o(s) = P_o(t)$ and $P_o(s') = P_o(t')$.

Example 2.13 Assume that G is the automaton represented in Figure 2.6, and let $\Sigma = \{a, b, c\}$, $\Sigma_o = \{a, b\}$, $\Sigma_{uo} = \{c\}$ and $X_s = \{3\}$. Then, we can say that G is 1-step opaque but not that it is 2-step opaque. In order to show that fact, let us consider automaton $O_{bs}(G, \Sigma_o)$ depicted in Figure 2.8. Note that, when the Intruder observes the sequence “ab”, it estimates states $\{3, 7\}$. Then, the only possible next observable sequence is “aba”, and now the Intruder knows that the system is currently in $\{4, 8\}$ and was in $\{3, 7\}$ one step back. Thus, the system is 1-step opaque. When the Intruder observes sequence “abaa”, it becomes sure that G is in state 4, therefore, if we consider automaton G shown in Figure 2.6, we see that the system was also in state 4 one step back and in state 3 two steps back. Thus the system is not 2-step opaque.

Note that CSO can be understood as 0-Step opacity, since the only concern of CSO is about the Intruder being sure that the current state is a secret one, *i.e.*, it takes into account zero past steps.

The last notion of state-based opacity, ∞ -step opacity, was introduced in SA-BOORI and HADJICOSTIS [9], and is an extension of K -Step Opacity. In this regard, a system is said to be ∞ -Step opaque if “the intruder is never sure whether the system has ever been in a secret state or not”. Its formal definition is as follows.

Definition 2.14 (∞ -Step Opacity) *Given a system modeled by $G = (X, \Sigma, f, \Gamma, x_0)$, projection $P_o : \Sigma^* \rightarrow \Sigma_o^*$, and a set of secret states $X_s \subseteq X$, G is ∞ -step opaque if for all $s \in L$, $f(x_0, s)!$ and for all $s' \in Pre(s)$ such that $f(x_0, s') \in X_s$, there exists $t \in L$, $f(x_0, t)!$, and $t' \in Pre(t)$ such that $f(x_0, t') \in X \setminus X_s$, $P_o(s) = P_o(t)$ and $P_o(s') = P_o(t')$.*

Example 2.14 *We use again the same automaton G represented in Figure 2.6, with the same sets $\Sigma = \{a, b, c\}$, $\Sigma_o = \{a, b\}$, $\Sigma_{uo} = \{c\}$ and $X_s = \{3\}$. Since the system is not 2-step opaque, as explained in Example 2.13, it will not be ∞ -step opaque. If we replace the self-loop in state 8 with another self-loop labeled with event “a”, then the system will become ∞ -step opaque, since the Intruder will never be sure if the system has ever been solely in the secret state 3.*

Notice that, even though Definition 2.8 concerns two different languages in a system, the notions of strong and weak opacity can be extended over Definitions 2.9 - 2.13. In this regard, the strong versions of LBO, CSO, ISO and IFO are defined by Definitions 2.9 - 2.12, whereas the weak versions will be obtained by replacing the restriction of “every secret” with “exists at least one secret” inside the definition; for example, we say that G is current-state weakly opaque with respect to Σ_o and X_s if $\exists s \in L$ such that $f(x_0, s) \in X_s$, and there exists $t \in L$ such that $f(x_0, t) \in X \setminus X_s$ and $P_o(s) = P_o(t)$. Finally, Definition 2.13 corresponds to the weak version of K -Step opacity, and the definition of K -Step Strong Opacity [21] is given by if the statement “for all $s \in L$, $f(x_0, s)!$ and for all $s' \in Pre(s)$ such that $|P(s) \setminus P(s')| \leq K$, $f(x_0, s') \in X_s$, there exists $t \in L$, $f(x_0, t)!$, $\forall t' \in Pre(t)$ such that $|P(t) \setminus P(t')| \leq K$, $f(x_0, t') \in X \setminus X_s$, $P_o(s) = P_o(t)$ and $P_o(s') = P_o(t')$ ” holds.

Example 2.15 *Assume again that G is the automaton represented in Figure 2.6, $\Sigma = \{a, b, c\}$, $\Sigma_o = \{a, b\}$, $\Sigma_{uo} = \{c\}$ but $X_s = \{3, 6\}$. Note that G is 1-step weakly opaque because whenever the Intruder estimates state 6, it also estimates states 1 and 2, no matter state 6 is being estimated as the current state or the 1-step back state. The same situation occurs when secret state 3 is estimated. It is not difficult*

to see that 1-Step Strong Opacity does not hold in G , since, after observing sequence “ab”, the intruder knows that, no matter what sequence has originally occurred, “acb” or “cab”, the system was in a secret state in at most 1 past steps. If the occurred sequence was “acb”, it means that the system is currently in secret state 3, and if the occurred sequence was “cab”, it means that the system has visited the secret state 6 in the 1-step back.

Note that the definition of these notions of opacity are used to characterize how “opaque” a system genuinely is, but when we deal with opacity-enforcement strategies, we always aim at the strong version of opacity. Thereafter, henceforth, whenever we talk about any opacity notion, we will refer to it as it was defined throughout this chapter, Definitions 2.9 - 2.14.

Since the current work deals with CSO enforcement, we need to know when the system satisfies this property. An easy and intuitive way to verify if a system is CSO is to build its observer automaton, since the state x_{obs} is understood to be the Intruder’s state estimation after observing a sequence $s \in L_{obs}$. Thus, we say that the system is CSO if all states of G_{obs} contains both secret and non secret states. Thereafter, the set of secret states of the observer is defined as $X_{s,obs} := X_{obs} \cap 2^{X_s}$. Thus, the system is CSO if and only if $X_{s,obs} = \emptyset$.

Chapter 3

Opacity-Enforcement methodology

This chapter is intended to expose the strategy developed to enforce current-state opacity through changes in the order of event observation. Section 3.1 characterizes the problem to be solved, the assumptions made on the model of the system, the capacity of the Intruder over the system and how the opacity-enforcer will work. Section 3.2 presents the opacity enforcement strategy itself and some definitions used to ease its comprehension. Section 3.3 presents the algorithms developed to enforce opacity through changes in the order of event observation. Finally, Section 3.4 show us an example where the strategy proposed in this work is applied.

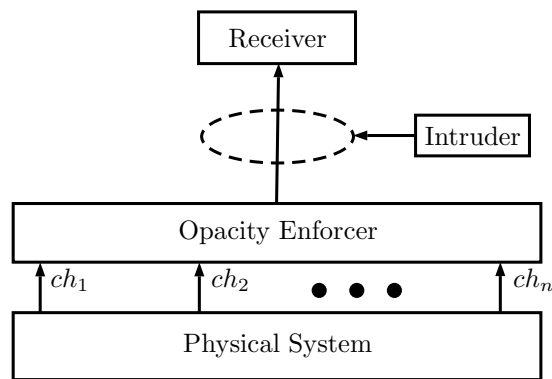


Figure 3.1: The Opacity-Enforcement architecture.

3.1 Problem Formulation

The architecture considered in this work is shown in Figure 3.1, and is composed of a physical system, which is modeled by an automaton, measurement points, which are responsible for recording the event occurrences, and transmission channels, which transmit the sensor signals to the Opacity-Enforcer. The Opacity-Enforcer is responsible for, after receiving an event, choosing, based on preset rules, either to release or hold it until the arrival of other events, with a view to changing the order of the observation of the events. The events released by the Opacity-Enforcer are transmitted to the intruder through a non-secure network.

With respect to the automaton model, only observable events are transmitted through the channels and although both, Receiver and Intruder, have the same power of observation based on the observable events, only the Receiver knows the rules used by the Opacity-Enforcer to enforce the desired opacity and the model. The Intruder, on the other hand, only has knowledge on the system model. The system secrets are represented by secret states, which the Intruder must never discover, *i.e.*, it can never estimate a secret state with certainty. The Opacity-Enforcer actions must prevent the Intruder from correctly estimating any secret state, supplying it with estimates of non-secret states.

The Opacity-Enforcer we propose here works as follows. When it receives a signal associated with an event occurrence, it may immediately release this signal or hold it until one or more events occur. When the Opacity-Enforcer holds the transmission of an event, it must release this event immediately after some sequence arrival in the sequel. To this end, the Opacity-Enforcer delays the event release by a certain number of steps, where step is understood here as any arrival of event at the Opacity-Enforcer.

We make the following assumptions on the Intruder's capacity:

- I1.** The Intruder has a copy of the automaton that models the system, including events, states, transition functions, as well as its initial states.

- I2.** The Intruder has full access to the signals transmitted from the Opacity-Enforcer to the Receiver, *i.e.*, it can observe all of the observable events.
- I3.** The Intruder does not know about the existence of the Opacity-Enforcer, and so, it does not know that the information might have been changed.
- I4.** The Intruder always expects to estimate some state inside the model whenever it observes an event.

We also make the following assumptions on the channels:

- A1.** Each channel that connects the physical system to the Opacity-Enforcer transmits only one specific event at time.
- A2.** There are no delays on transmission through the communication channels.
- A3.** The channels connecting the physical system and the Opacity-Enforcer are safe.
- A4.** The network connecting the Opacity-Enforcer and the Receiver is unsafe.

Assumption A1 means that channel ch_1 will just transmit an event σ_1 to the Opacity-Enforcer, ch_2 will just transmit an event σ_2 and so on. Assumption A2 means that the dynamic of the system is much slower than the transmission of the events through the channels, *i.e.*, the transmission delays are negligible. Assumptions A3 and A4 mean that only the transmission from the Opacity-Enforcer to the Receiver is susceptible to leak information to the Intruder.

3.2 The Opacity-Enforcement Strategy

The strategy proposed here leverages the possibility of delaying events so as to cause changes in its order of observation, misleading the Intruder's estimation of the current state of the system. To this end, the Opacity-Enforcer checks if for every sequence that reaches a secret state with respect to the observer and its continuations, there exists some possible change in the order of observation of the

events so that this sequence can be seen as another possible one whose prefixes and their continuations lead only to non-secret states of the observer. The changes in the order of event observation is a permutation of the order of the events that occurs in a sequence. Although shuffling the event order in a sequence is an easy task, its formal definition requires some effort. First we need to consider the sequence as an ordered tuple; then, we arrange randomly the elements of the tuple, and; finally, we concatenate the elements of the randomly arranged tuples in sequences again. We start by defining the function “Tuple of a sequence”, which maps a sequence into an ordered tuple, and its inverse mapping.

Definition 3.1 (Tuple of a sequence) *Let $s = \sigma_1\sigma_2\dots\sigma_n \in \Sigma^*$ be a finite sequence. The tuple of a sequence s is the mapping $T : \Sigma^* \rightarrow \Sigma^n$, where $n = |s|$, $\Sigma^n = \Sigma \times \Sigma \cdots \times \Sigma$, n times, is defined as the n -tuple $T(s) := (\sigma_1, \sigma_2, \dots, \sigma_n)$ formed by the events of s .*

Giving the n -tuple $t = (t_1, t_2, \dots, t_n) \in \Sigma^n$, the inverse mapping $T^{-1} : \Sigma^n \rightarrow \Sigma^$ is defined as the sequence $T^{-1}(t) := t_1t_2\dots t_n$ formed by the concatenation of the elements in the n -tuple t , according to the order they appear in t .*

Note that the operation sequence from a tuple can be extended to a set of tuples $T^{-1} : 2^{\Sigma^n} \rightarrow 2^{\Sigma^*}$ such that $T^{-1}(A) := \{s \in \Sigma^* : (\exists t \in A)[s = T^{-1}(t)]\}$.

The next operation that must be defined is the “Permutation of tuples”, which, given a tuple t as input, generates all possible tuples t' such that the order of the elements in t' is a permutation of the elements in t .

Definition 3.2 (Permutation of tuples) *Let $t = (t_1, t_2, \dots, t_n) \in \Sigma^n$. The permutation of a tuple is the mapping $\mathcal{P} : \Sigma^n \rightarrow 2^{\Sigma^n}$, that associates to each n -tuple t , a set $\mathcal{P}(t)$ of n -tuples composed of all permutations of the order in the elements of it, i.e., $\mathcal{P}(t) := \{(t_1, t_2, \dots, t_{n-1}, t_n), (t_1, t_2, \dots, t_n, t_{n-1}), \dots, (t_n, t_{n-1}, \dots, t_2, t_1)\}$.*

Notice that, given an n -tuple $t = (t_1, t_2, \dots, t_n)$, $|\mathcal{P}(t)| \leq n!$, since it is possible that there exists $i, j \in \{1, 2, \dots, n\}$, $i \neq j$.

Finally, we define the “Sequence permutation”, as the mapping that generates all event order permutations of a given sequence.

Definition 3.3 (Sequence permutation) *Let $s = \sigma_1\sigma_2\dots\sigma_n \in \Sigma^*$ be a sequence and $T(s) = (\sigma_1, \sigma_2, \dots, \sigma_n)$ denote the n -tuple formed from that sequence. The sequence permutation S_p is a mapping $S_p : \Sigma^* \rightarrow 2^{\Sigma^*}$ where for each sequence $s = \sigma_1\sigma_2\dots\sigma_n$, it associates a set $S_p(s) := T^{-1}(\mathcal{P}(T(s)))$.*

The following example illustrates the operations presented in Definitions 3.1 – 3.3.

Example 3.1 *Let us consider the sequence $s = aba$. We have that $t = T(s) = (a, b, a)$ is the corresponding tuple of s . The permutations defined from $T(s)$ form the set $\mathcal{P}(t) = \{(a, b, a), (a, a, b), (b, a, a)\}$ and thus, the sequence permutation of s is $S_p(s) = T^{-1}(\mathcal{P}(t)) = \{aba, aab, baa\}$. Notice that, as pointed out before, $|\mathcal{P}(t)| = 3 < 3! = 6$.*

Now, that the sequence permutation has been defined, the next step in the opacity enforcement strategy proposed in this work is to search for conditions that ensure the feasibility of the proposed opacity enforcement, *i.e.*, the conditions in order for the system to be opacity-enforceable.

Definition 3.4 (Opacity-enforceability) *A system modeled by an automaton $G = (X, \Sigma, f, \Gamma, x_0)$ is CSO enforceable through changes in the order of event observations with respect to Σ_o and X_s if $\forall s : f_{obs}(x_{0,obs}, s) \in X_{s,obs}$, and $\forall t \in L_{obs}/s, \exists n \in \mathbb{Z}_+$ and $\exists u \in L_{obs}/st : |u| = n$, and $\exists v \in S_p(stu) \cap L_{obs}$ such that $f_{obs}(x_{0,obs}, w) \in X_{obs} \setminus X_{s,obs}, \forall w \in Pre(v)$.*

Notice that, in Definition 3.4, n can be equal to 0.

Example 3.2 *Let us consider automaton G_1 , whose state transition diagram is depicted in Figure 3.2a, and let $\Sigma_o = \{a, b, c, d\}$ and $X_s = \{2\}$. Note that its observer automaton $G_{obs,1}$ is identical to G_1 . According to Definition 3.4, the system*

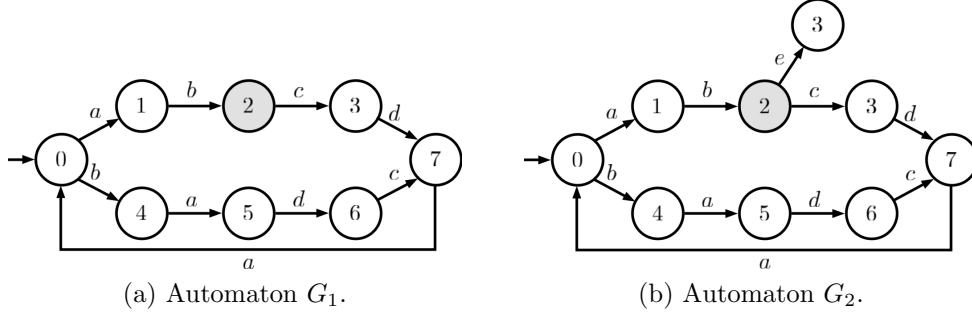


Figure 3.2: Automata used in Examples 3.2.

modeled by automaton G_1 is CSO enforceable. In order to show this fact, notice that for $s = ab$, $f_{obs}(x_{0,obs}, s) = 2$. Let us now consider all continuations of L_{obs} after s .

- $t = \varepsilon$. In this case, there exists an integer $n = 2$ and a sequence $u = cd \in L_{obs}/st$ such that $v = badc \in S_p(stu) \cap L_{obs,1} = \{abcd, badc\}$ and $f_{obs,1}(0, w) \notin X_{s,obs,1}, \forall w \in Pre(v) = \{\varepsilon, b, ba, bad, badc\}$;
- $t = c$. In this case, there exists $n = 1$ and $u = d$, such that Definition 3.4 holds, since the same $v = badc$ is obtained here;
- Continuing this process for $t = cd$, $t = cda$, $t = cdaa, \dots$, similar conclusions can be drawn.

Let us now consider the system modeled by automaton G_2 , depicted in Figure 3.2b, where $\Sigma_o = \{a, b, c, d, e\}$ and $X_s = \{2\}$. Notice that this system is not CSO enforceable, since for $s = ab$ and $t = e$, there does not exist any integer n , any sequence u after st in L_{obs} , and, hence, any $v \in S_p(stu) \cap L_{obs,2}$ such that $f_{obs,2}(0, w) \notin X_{s,obs,2}, \forall w \in Pre(v)$

In order to define an opacity enforcement strategy, the occurrence of the event in the system must be distinguished from its observation by the Intruder. To this end, let Σ_s be a copy of Σ with all of its events relabeled by introducing a subscript s . We define the following operation to rename the events of a sequence $s \in \Sigma^*$.

Definition 3.5 (Rename Operation) *The rename operation is the mapping $R : \Sigma^* \rightarrow \Sigma_s^*$, where: $R(\varepsilon) = \varepsilon$, $R(\sigma) = \sigma_s$ and $R(s\sigma) = (s\sigma)_s = R(s)R(\sigma) = R(s)\sigma_s$,*

where $\sigma \in \Sigma$ and $s \in \Sigma^*$. The inverse rename operation $R^{-1} : \Sigma_s^* \rightarrow \Sigma^*$ can be defined as $R^{-1}(\varepsilon) = \varepsilon$, $R^{-1}(\sigma_s) = \sigma$ and $R^{-1}((s\sigma)_s) = R^{-1}(s_s\sigma_s) = R^{-1}(s_s)R^{-1}(\sigma_s) = R^{-1}(s_s)\sigma$.

Based on the rename operation, we define the set of actual event observations as follows:

$$\Sigma_{o,s} := \{\sigma_s : (\sigma_s = R(\sigma)) \wedge (\sigma \in \Sigma_o)\} \quad (3.1)$$

We then define the extended set of events as $\Sigma_e := \Sigma_o \cup \Sigma_{o,s}$.

Assume that opacity can be enforced in a non-opaque system through changes in the order of event observation. Thus, there exists an opacity enforcement strategy OE such that for every sequence that has occurred, it decides, after each event arrival, whether: (i) it releases immediately its observation; (ii) it holds it without releasing any observation at all or; (iii) it holds the event but releases the observation of another event that has been held. In order to present a formal definition for the opacity enforcement function, let us define the function $\mathcal{N} : \Sigma_e^* \times \Sigma_e \rightarrow \mathbb{Z}_+$, where for a pair (s, σ) , $\mathcal{N}(s, \sigma)$ returns the number of occurrences of event σ in sequence s .

Definition 3.6 (Opacity-Enforcement function) *Let us consider the projections $P_{e,s} : \Sigma_e^* \rightarrow \Sigma_{o,s}^*$. The opacity enforcement function $OE : \Sigma_e^* \rightarrow 2^{\Sigma_{o,s}} \cup \{\varepsilon\}$ is defined as:*

$$OE(s_e) := \begin{cases} \sigma_s \in \Sigma_{o,s}, & \text{if } \mathcal{N}(s_e, \sigma_s) < \mathcal{N}(s_e, R^{-1}(\sigma_s)) \wedge \\ & f_{obs}(x_{0,obs}, R^{-1}(P_{e,s}(s_e)\sigma_s)) \in X_{obs} \setminus X_{s,obs} \\ \varepsilon, & \text{otherwise} \end{cases}$$

Note that, by imposing that $\mathcal{N}(s_e, \sigma_s) < \mathcal{N}(s_e, R^{-1}(\sigma_s))$, we ensure that only events that have actually occurred can be released for observation, as opposed to the insertion function used in [17–19] that allows the introduction of fictitious events. In addition, the released observation must lead to non-secret states of the observer, which implies that under no circumstances the intruder will be in doubt if the system

is in a secret or non-secret state of G_{obs} , but always thinks that the system is in a non-secret state of G_{obs} .

Example 3.3 *Let us consider automaton G_1 depicted in Figure 3.2a, and let $\Sigma_o = \{a, b, c, d\}$ and $X_s = \{2\}$. Initially, the opacity enforcement function will return $OE(\varepsilon) = \varepsilon$ and wait for some event occurrence. Upon the occurrence of $s_e = a$, it will release no event, $OE(a) = \varepsilon$. Notice that, the release of a_s would eventually lead the Opacity-Enforcer to hold indefinitely every further occurred event, since all of the next events would either make the intruder to estimate the secret state 2 or generate a sequence not in the observable behavior of the system. After releasing no observation when “a” occurred, the function will wait for “b” to occur and release “b_s”, $OE(ab) = b_s$. Then, the next action of the enforcer function is to release the held event “a_s”, $OE(abb_s) = a_s$, leading the Intruder to estimate state 5 while the system is in 2. Carrying out this procedure, we have that $OE(abb_s a_s) = \varepsilon$, i.e., the enforcer is waiting for some event occurrence, $OE(abb_s a_s c) = \varepsilon$, $OE(abb_s a_s cd) = d_s$, $OE(abb_s a_s cdd_s c_s) = \varepsilon$, and so on. Notice that while sequence abcd occurred in the system, the opacity enforcement function released $b_s a_s d_s c_s$, misleading the Intruder to wrongly estimate states.*

When we embed the opacity-enforcement function into the physical system, the resulting sequences s_e , obtained by the action of the opacity enforcer, generate an extended language L_e that depends on the language generated by the system and the event observations released by the opacity enforcer. This language can be recursively defined as follows.

1. $\varepsilon \in L_e$, since $OE(\varepsilon) = \varepsilon$ and thus, $s_e = \varepsilon \in L_e$
2. $s_e \sigma \in L_e \iff (s_e \in L_e) \wedge (P_{e,o}(s_e \sigma) \in L_{obs})$
3. $s_e \sigma_s \in L_e \iff (s_e \in L_e) \wedge (\sigma_s \in OE(s_e))$

where $\sigma \in \Sigma_o$ is an event occurrence and $\sigma_s \in \Sigma_{o,s} \cup \{\varepsilon\}$ is either the empty sequence ε or the event released by the opacity enforcer.

Note that, according to the definition of L_e , at least one new sequence is added to the extended language every time an observable event σ occurs in G . If the Opacity-Enforcer decides not to release any new observation, then only sequences of type $s_e\sigma$ are added to L_e . However, when the Opacity-Enforcer releases an event observation $\sigma_s \in \Sigma_{o,s}$ after s_e , then $s_e\sigma_s$ is also added to L_e .

It is straightforward to see from the definition of extended language that $P_{e,o}(L_e) = L_{obs}$, where $P_{e,o} : \Sigma_e^* \rightarrow \Sigma_o^*$ and that $\forall s_e \in L_e, f_{obs}(x_{0,obs}, R^{-1}(P_{e,s}(s_e))) \in X_{obs} \setminus X_{s,obs}$.

Note that $|OE(s_e)| > 1$ occurs when it is possible for the observation of two or more events to lead to different non-secret states of G_{obs} . In this case, some event release policy must be adopted; the most intuitive one is to release the event that has happened first. If $OE(s_e) = \varepsilon$, then, no event will be released, which suggests a trivial way to enforce CSO, *i.e.*, by holding all events observations, *i.e.*, $OE(s_e) = \varepsilon, \forall \sigma_e$ and, eventually, if some σ_s appears such that it is possible for G_{obs} to move to a non-secret state, then $OE(s_e)$ releases it. Such a solution is not wise from the practical point of view, and thus, it is more realistic to establish bounds on the number of steps that observable events can be held. In this regard, the Opacity-Enforcer actions will be bounded in order to make it feasible, *i.e.*, they will be bounded by a maximum step delay associated with each event. This Opacity-Enforcer property is described as

$$SD = \{(\sigma_1, k_1), (\sigma_2, k_2), \dots, (\sigma_n, k_n)\} \quad (3.2)$$

where $k_i, i = 1 \dots n$, represents the maximum number of steps that event σ_i can be held. For example, assuming that $SD = \{(\alpha, 1), (\beta, 0), (\gamma, 0)\}$, then if the sequence $\alpha\beta\gamma$ arrives, the Opacity-Enforcer may release α immediately, creating the shuffled sequence $\alpha\alpha_s\beta\beta_s\gamma\gamma_s$, or hold it for one step, creating the shuffled sequence $\alpha\beta\beta_s\alpha_s\gamma\gamma_s$; thus two output observation sequences, $\alpha_s\beta_s\gamma_s$ or $\beta_s\alpha_s\gamma_s$ can be chosen depending on the CSO enforcement policy. With this constraint, we present the following necessary and sufficient condition for CSO enforceability.

Theorem 3.1 Let $P_{e,o} : \Sigma_e^* \rightarrow \Sigma_o^*$. We say that $G = (X, \Sigma, f, \Gamma, x_0)$ is CSO enforceable through changes in the order of event observations with respect to Σ_o , X_s and SD if and only if $\forall s : f_{obs}(x_{0,obs}, s) \in X_{s,obs}$, and $\forall t \in L_{obs}/s, \exists n \in \mathbb{Z}_+, \exists u \in L_{obs}/st : |u| = n, \exists s_e \in L_e$ such that: (i) $P_{e,o}(s_e) = stu$; (ii) $\forall \sigma \in \Sigma_o, \mathcal{N}(s_e, \sigma) = \mathcal{N}(s_e, R(\sigma))$; and (iii) SD holds true for all observable events.

Proof: Let $P_{e,s} : \Sigma_e^* \rightarrow \Sigma_{o,s}^*$.

(\Rightarrow) We will prove the sufficient condition by taking the contrapositive of the implication. Thus, assume that $\exists s : f_{obs}(x_{0,obs}, s) \in X_{s,obs}$, and $\exists t \in L_{obs}/s, \forall n \in \mathbb{Z}_+, \forall u \in L_{obs}/st : |u| = n, \forall s_e \in L_e$ such that either conditions (i), (ii) or (iii) is false. Let us consider an extended sequence $s_e \in L_e$ such that conditions (i) and (ii) hold but (iii) does not hold. It is straightforward to see that if condition (iii) is not met, then the system is not CSO enforceable with respect to SD . If conditions (i) and (iii) are met but not condition (ii), then $\forall s_e \in L_e$ we have that $(P_{e,o}(s_e) = stu) \wedge (\mathcal{N}(s_e, \sigma) \neq \mathcal{N}(s_e, R(\sigma))) \wedge (SD \text{ holds})$, thus, by setting $v = R^{-1}(P_{e,s}(s_e))$, we have that $v \notin S_p(stu)$, since $|v| < |stu|$, hence, $\nexists v \in S_p(stu) \cap L_{obs}$ such that $f_{obs}(x_{0,obs}, w) \in X_{obs} \setminus X_{s,obs}, \forall w \in Pre(v)$, leading us to conclude that, according to Definition 3.4, the system is not CSO enforceable. Finally, condition (i) being false is not feasible, since $P_{e,o}(L_e) = L_{obs}$, thus, there always exists $s_e \in L_e$ such that $P_{e,o}(s_e) = stu \in L_{obs}$. Hence, according to Definition 3.4, the system is not CSO enforceable.

(\Leftarrow) Assume that $\forall s : f_{obs}(x_{0,obs}, s) \in X_{s,obs}$, and $\forall t \in L_{obs}/s, \exists n \in \mathbb{Z}_+$ and $\exists u \in L_{obs}/st : |u| = n$, and $\exists s_e \in L_e$ such that conditions (i) – (iii) hold. Let us set $v = R^{-1}(P_{e,s}(s_e))$. Thus, since conditions (i) and (ii) are met and s_e is an extended sequence of the language L_e , obtained through the actions of an opacity enforcement function, we can say that $v \in S_p(stu)$ and that $v \in L_{obs}$, and, in addition, that $f_{obs}(x_{0,obs}, w) \in X_{obs} \setminus X_{s,obs}, \forall w \in Pre(v)$. Hence, we can write our assumption as $\forall s : f_{obs}(x_{0,obs}, s) \in X_{s,obs}$, and $\forall t \in L_{obs}/s, \exists n \in \mathbb{Z}_+, \exists u \in L_{obs}/st : |u| = n, \exists v \in S_p(stu) \cap L_{obs}$ such that $f_{obs}(x_{0,obs}, w) \in X_{obs} \setminus X_{s,obs}, \forall w \in Pre(v)$, then, from Definition 3.4, the system is CSO enforceable with respect to Σ_o, X_s and SD , since

condition (iii) is also met. ■

3.3 Algorithms

In this section, we first adapt the algorithm proposed in [29] in Subsection 3.3.1. In Subsection 3.3.2, we propose an algorithm which checks whether CSO enforcement with respect to Σ_o , X_s and SD can be done or not. Finally, in Subsection 3.3.3, we propose an algorithm that searches for a minimum delay bounds such that the system CSO enforceable. Both, the two proposed algorithms, are jointly used to solve the opacity enforcement problem. These algorithms do not deal with decision conflict problems when $|OE(s_e)| > 1$, *i.e.*, when two or more events are eligible to be released by the opacity enforcement function.

The shuffle of event observations is carried out by constructing automaton D , which was first presented in [29] to model the effect of communication delay in language diagnosability. Here, automaton D will be used with a different purpose, *i.e.*, to synchronize *all possible changes in the order of observation* according to the opacity-enforcement property such that the *altered sequence observations remain inside the language generated by the system*.

Assuming that an upper bound is given, the first proposed algorithm (*Algorithm 3.3*) finds a minimal solution with respect to how long each event must be held in order to enforce opacity. The second proposed algorithm (*Algorithm 3.2*) is executed until either a minimal solution is found or opacity cannot be enforced with the current bound SD . In addition, *Algorithm 3.2* makes the distinction between event occurrences and observations, and although it shuffles the event observation order, it preserves the language generated by the system, *i.e.*, *Algorithm 3.2* ensures that for every sequence that reaches a secret state and its continuations, there must exist some change in the order of its event observations that leads only to estimations of non-secret states of G_{obs} by the Intruder.

3.3.1 Computation of automaton D

Automaton D_i was conceived in the context of failure diagnosis for communication delay. This architecture has the following components: *(i)* a Physical System with Measurement Sites (MS_j), from where the recorded events are transmitted; *(ii)* Local Diagnosers (LD_i), which fire when a failure is detected, and; *(iii)* channels ($ch_{i,j}$), which follow FIFO rule, transmit one event at a time from the measurement sites to the local diagnosers, and are susceptible to a known delay. Given the event set $\Sigma = \Sigma_o \cup \Sigma_{uo}$ with respect to the system, a vector of delays $k_{i,j}$ with respect to each channel, and the set of events $\Sigma_{i,j}$ transmitted through them, automaton D_i generates all sequences with respect to Σ_o and also all possible changes in the order of event observation due to delays, with respect to $\Sigma_{o,s}$. The states of D_i denote occurred sequences which possesses events not successfully observed yet. The symbol ν , which represents “*blank space*”, is also used when naming the states of D_i , since a successful observation of an event that is not its first occurred event or the occurrence of unobservable events leads to new states with blank spaces in their names. This blank space ν must be considered in order to control how many steps the event observations are being delayed. Note that the initial state of D_i is also denoted as ν , since no event has occurred.

We make some simplifications over automaton D , since we do not deal with either measurement sites or multiple local Opacity-Enforcers. Note that, in our context, instead of local diagnosers, we have a single local receiver, namely the Opacity Enforcer. Automaton D_i is adapted to our context by assuming that we have one MS for each event and also that the associated channel to this MS transmits this specific event only, as stated in assumption **A1**, thus, every event can have its observation changed with another event observation, as long as the delay associated to them is enough to enable that exchange. We also deal with one Opacity Enforcer only, leading to the construction of one unique automaton D instead of multiple automata D_i . Some steps of the original construction algorithm are negligible, since we compute D with respect to the event set of the observer automaton, *i.e.*, there

are no unobservable events to be considered.

Before we proceed to the computation of automaton D , some operations over sequences $s \in (\Sigma_o \cup \{\nu\})^*$, which are used in order to build D , must be defined.

Definition 3.7 *Let us define $\Sigma_{o\nu} = \Sigma_o \cup \{\nu\}$ and the set of states Q , where each state $q \in Q$ is labeled with a sequence $s \in \Sigma_{o\nu}^*$. Then, given $\Sigma = \Sigma_o \cup \Sigma_{uo}$, we define the following functions:*

- *Replacement function, defined as $rep : Q \times \mathbb{Z}_+^* \rightarrow Q$, where $\forall q = q_1q_2 \dots q_l \in Q$ and $\mathbb{Z}_+^* = \mathbb{Z}_+ \setminus \{0\}$,*

$$rep(q, i) = \begin{cases} q_1q_2 \dots q_{i-1}\nu q_{i+1} \dots q_l, & \text{if } i \leq l \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

- *Elimination function, defined as $cut : Q \rightarrow Q$, where $\forall q = q_1q_2 \dots q_l \in Q$,*

$$cut(q) = \begin{cases} q_iq_{i+1} \dots q_l, & \text{if } (\exists i \leq l)[(q_i \neq \nu) \wedge (q_k, \forall k \in \{1, 2, \dots, i-1\})] \\ \nu, & \text{if } q_k = \nu, \forall k \in \{1, 2, \dots, l\}. \end{cases}$$

Example 3.4 *Let us consider a state $q = ab$, $a \in Q$. We have that $cut(rep(q, 1)) = cut(\nu b) = b$ and that $cut(rep(q, 2)) = cut(a\nu) = a\nu$.*

Since we have already simplified the structure of automaton D and defined the operations required to its construction, we now present the algorithm which computes automaton D .

Algorithm 3.1 *Computation of automaton D*

Input: $\Sigma_o, k^{max} = [k_{\sigma_1}^{max}, \dots, k_{\sigma_n}^{max}]$.

Output: $D = (X_D, \Sigma_e, f_D, \Gamma_D, x_{0,D})$.

STEP 1. *Define $x_{0,D} = \nu$ and $X_D = \emptyset$.*

STEP 2. *Construct $\Sigma_{o,s} = R(\Sigma_o)$ and then define $\Sigma_e = \Sigma_o \cup \Sigma_{o,s}$.*

STEP 3. $F \leftarrow x_{0,D}$, where F denotes a FIFO queue.

STEP 4. While $F \neq \emptyset$, do:

STEP 4.1. $u \leftarrow \text{head}[F]$

STEP 4.2. If $u = x_{0,D}$, then:

STEP 4.2.1. For each $\sigma \in \Sigma_o$, compute $\tilde{x}_D = f_D(u, \sigma) = \sigma$, then $\text{Enqueue}(F, \tilde{x}_D)$.

STEP 4.2.2. $X_D \leftarrow X_D \cup \{u\}$

STEP 4.2.3. $\text{Dequeue}(F)$

Else:

STEP 4.2.4. Set $\ell = |u|$ and create the set $I_\ell = \{1, 2, \dots, \ell\}$

STEP 4.2.5. Denote $u = \sigma_1\sigma_2\dots\sigma_\ell$ and create the set $I_\nu = \{y \in I_\ell :$

$$(\exists \sigma_y \in u)[\sigma_y = \nu]\}$$

STEP 4.2.6. Compute $I_{\ell \setminus \nu} = I_\ell \setminus I_\nu$

STEP 4.2.7. For each $\sigma \in \Sigma_o$, do:

$$(a) \tilde{x}_D = f_D(u, \sigma) = \begin{cases} u\sigma, & \text{if } |\sigma_y\sigma_{y+1}\dots\sigma_\ell| \leq k_{\sigma_y}^{max}, \forall y \in I_{\ell \setminus \nu} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

(b) If \tilde{x}_D is defined, then $\text{Enqueue}(F, \tilde{x}_D)$

STEP 4.2.8. For each $\sigma_s \in \Sigma_{o,s}$, do:

(a) Create the set $Y = \{y : (\sigma_y \in u) \wedge (\sigma_y = R^{-1}(\sigma_s))\}$

(b) If $Y \neq \emptyset$, then compute $\hat{y} = \min(Y)$ and $\tilde{x}_D = f_D(u, \sigma_s) = \text{cut}(\text{rep}(u, \hat{y}))$

(c) If $(\tilde{x}_D \notin X_D) \wedge (\tilde{x}_D \notin F)$, then $\text{Enqueue}(F, \tilde{x}_D)$

STEP 4.2.9. Set $X_D \leftarrow X_D \cup \{u\}$

STEP 4.2.10. $\text{Dequeue}(F)$

STEP 5. For each $x_D \in X_D$, $\Gamma_D(x_D) = \{\sigma \in \Sigma_e : f_D(x_D, \sigma)!\}$

We start *Algorithm 3.1* by creating the initial state $x_{0,D}$ in STEP 1, naming it ν , since no event have occurred yet, and by setting the set of states X_D as an empty

set. In STEP 2, we build $\Sigma_{o,s}$ labeling each event of Σ_o with subscript s by using the rename function $R(\sigma)$, and then, we build the extended set of events Σ_e , which is composed by all of the occurred and observed events. In STEP 3, we create a FIFO queue named F and set its first element as the initial state of D . In STEP 4, we enter a routine where we will analyze all possible event occurrences and event successful observations, that may lead to the creation of new transitions and states. The states in automaton D are named after the observed sequences that reaches them whose event observations have not been released yet, for example: if state x_i is reached by sequence ac , then $x_i = ac$, and; if a state x_j is reached by sequence acc_s , then the state reached by this sequence will be $x_j = av$. Note that we will repeat STEP 4 until the queue F is empty. In STEP 4.1 we take the first state u of the queue F and check, in STEP 4.2, whether it is the initial state or not. If we are analyzing the initial state, then we proceed to STEP 4.2.1, where, for each observable event $\sigma \in \Sigma_o$, we create a state $\tilde{x}_D = \sigma$ and also the transition function $f_D(\nu, \sigma) = \sigma$, and then, we add these states, represented as \tilde{x}_D , to the queue F . In STEP 4.2.2, we add the analyzed state u to the set of states X_D , and then, in STEP 4.2.3, we remove the analyzed state u from the queue F . If the current state u , which we are analyzing, is not the initial state, we proceed to STEP 4.2.4, where we measure the length of the sequence that reaches u , $\ell = |u|$, and then, we create a set whose elements are the positive integers smaller or equal to ℓ , *i.e.*, $I_\ell = \{1, 2, \dots, \ell\}$. In STEP 4.2.5, we denote u as a sequence of events indexed with ascending numbers, $u = \sigma_1\sigma_2 \dots \sigma_\ell$, and then, we create a set I_ν containing all integers representing the order the “blank space” ν has appeared in $u = \sigma_1\sigma_2 \dots \sigma_n$. In STEP 4.2.6, we create the set $I_{\ell \setminus \nu}$ as the set difference between I_ℓ and I_ν , *i.e.*, $I_{\ell \setminus \nu} = I_\ell \setminus I_\nu$, representing all indexed numbers of events that still have to be observed. In STEP 4.2.7, which concerns the event occurrences with respect to their maximum delay value, we create new states $\tilde{x}_D = u\sigma$ and transition functions $f_D(u, \sigma) = u\sigma$ for every observable event $\sigma \in \Sigma_o$, as long as the number of allowed delay of each element of u , events yet to be observed, has not surpassed its maximum value, $|\sigma_y\sigma_{y+1} \dots \sigma_\ell| \leq k_{\sigma_y}^{max}$, and

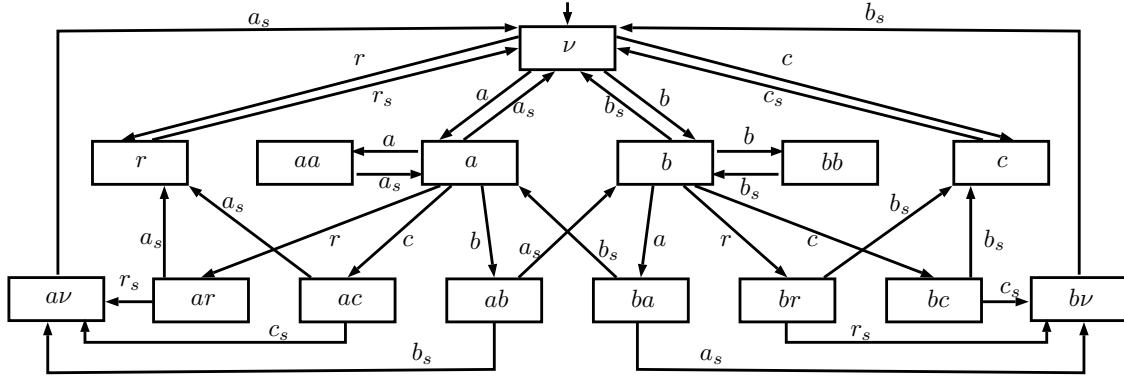


Figure 3.3: Automaton D .

so, if \tilde{x}_D is defined, we add it to the queue F . In STEP 4.2.8, we deal with the successful event observations, and, for each observed event $\sigma_s \in \Sigma_{o,s}$, we create the set Y containing all index numbers of the elements of u which corresponds to the occurrence of σ_y , *i.e.*, $\sigma_y = R^{-1}(\sigma_s)$, and so, if this set is not empty, $Y \neq \emptyset$, then we take the least element of Y , $\hat{y} = \min(Y)$ and then replace the first appearance of event σ_y , which is denoted as $\sigma_{\hat{y}}$, with the “blank space” ν . After that, we erase all elements ν of u which are not preceded by any event and set it as new state, *i.e.*, we set $\tilde{x}_D = \text{cut}(\text{rep}(u, \hat{y}))$. We also define the transition $f_D(u, \sigma_s) = \tilde{x}_D$ and add \tilde{x}_D to F if it has neither been analyzed nor it already belongs to F . In STEP 4.2.9, we add u to the set of states X_D and, in STEP 4.2.10, we remove u from the queue F . Finally, in STEP 5 we compute the set of active events of automaton D .

Note that all elements of the queue F are states yet to be analyzed, while the elements of the set X_D are states which have already been through Algorithm 3.1 as the head element u , *i.e.*, they already were analyzed.

Example 3.5 Given $\Sigma_o = \{a, b, c, r\}$, and $k^{\max} = [k_a^{\max}, k_b^{\max}, k_c^{\max}, k_r^{\max}] = [1, 1, 0, 0]$, we will now build automaton D , whose state transition diagram is depicted in Figure 3.3, following along Algorithm 3.1. In STEP 1, we create the initial state $x_{0,D} = \nu$ and set $X_D = \emptyset$. In STEP 2, we set $\Sigma_{o,s} = R(\Sigma_o) = \{a_s, b_s, c_s, r_s\}$ and $\Sigma_e = \Sigma_o \cup \Sigma_{o,s} = \{a, a_s, b, b_s, c, c_s, r, r_s\}$. In STEP 3, we create the queue $F = [\nu]$. In STEP 4, we start building D from its initial state ν ($u = \nu$), and, since no event has occurred yet, we create states for all possible event occurrences, *i.e.*, events “a”,

“b”, “c” and “r” can occur, and so, states “a”, “b”, “c” and “r” are created with their correspondent transitions; after that, we add these new states to the queue, i.e., $F = [\nu, a, b, c, d, r]$, update the state set to $X_D = \{\nu\}$ and remove the head element ν from the queue, as stated in STEPS 4.2.1 – 4.2.3. The next element of the queue F is state “a” ($u = a$), therefore, we set $\ell = 1$, $I_\ell = \{1\}$, $I_\nu = \emptyset$, $I_{\ell \setminus \nu} = \{1\}$ in STEPS 4.2.4 – 4.2.6; since $|a| \leq k_a^{\max} = 1$, all events are enabled to occur in state “a”, and, thus, in STEP 4.2.7, states “aa”, “ab”, “ac” and “ar” are created in D and added to the queue $F = [a, b, c, d, r, aa, ab, ac, ar]$; in STEP 4.2.8, we have one iteration for a_s only, where we set $Y = \{1\}$, $\hat{y} = 1$ and add the transition $f_D(a, a_s) = \text{cut}(\text{rep}(a, 1)) = \text{cut}(\nu) = \nu$ to D . We set $X_D = \{\nu, a\}$ and remove the head element, state “a”, from queue F in STEPS 4.2.9 and 4.2.10, respectively. We carry out this process for all states in queue F until it becomes empty.

Notice that, when we analyze state “ab” ($u = ab$), we have that $|ab| = 2 \not\leq k_a^{\max} = 1$, thus, no states are created in STEP 4.2.7, and so, we proceed to STEP 4.2.8, where we set $\sigma_s = a_s$, and, hence, we have that $Y = \{1\}$, $\hat{y} = 1$ and transition $f_D(ab, a_s) = \text{cut}(\text{rep}(ab, 1)) = \text{cut}(\nu b) = b$ is added to D ; in the second iteration, where $\sigma_s = b_s$, we set $Y = \{2\}$, $\hat{y} = 2$ and add the transition $f_D(ab, b_s) = \text{cut}(\text{rep}(ab, 2)) = \text{cut}(a\nu) = a\nu$ and the corresponding state $\tilde{x}_D = a\nu$ to automaton D .

3.3.2 An Algorithm for Opacity Enforcement Checking

Now that the construction of automaton D has been explained, we will introduce the first algorithm related to the opacity-enforcement strategy proposed here.

Given the observer automaton G_{obs} , a secret state set X_s and a step delay configuration SD , this algorithm investigates whether CSO can be enforced or not and also realizes the Opacity-Enforcer automaton R_{oe} , as follows.

Algorithm 3.2 *Opacity enforcement checking*

Input: G_{obs} , X_s , SD .

Output: (True, R_{oe}) or False.

STEP 1. Set $G_{sys} = (X_{sys}, \Sigma_o, f_{sys}, \Gamma_{sys}, x_{0,sys}) = G_{obs}$

STEP 2. Set $G_{int} = (X_{sys}, R(\Sigma_o), f_{int}, \Gamma_{int}, x_{0,sys})$, where $f_{int}(x, \sigma_s) = f_{sys}(x, \sigma)$ and $\Gamma_{int}(x) = R(\Gamma_{sys}(x))$

STEP 3. Construct automaton $D = (X_D, \Sigma_e, f_D, \Gamma_D, x_{0,D})$ as in Algorithm 3.1 using the property SD as input

STEP 4. Construct $V := G_{sys} || D || G_{int} = (X_V, \Sigma_e, f_V, \Gamma_V, x_{0,V})$

STEP 5. For each $x_V = (x_{sys}, x_D, x_{int}) \in X_V$:

(i) If $x_{int} \in 2^{X_s}$, then: delete x_V , set $V = Ac(V)$ and start STEP 5 again

(ii) If $x_D \neq x_{0,D}$ and $\Gamma_V(x_V) = \emptyset$, then: delete x_V , set $V = Ac(V)$ and start STEP 5 again

(iii) If $[\Gamma_{sys}(x_{sys}) \cap \Gamma_D(x_D)] \setminus \Gamma_V(x_V) \neq \emptyset$, then: delete x_V , set $V = Ac(V)$ and start STEP 5 again

STEP 6. For each $x_V \in X_V$:

STEP 6.1. If $\Gamma_V(x_V) \cap \Sigma_s \neq \emptyset$, then remove all events $\sigma \in \Sigma_e \setminus \Sigma_s$ from it and set $V = Ac(V)$

STEP 7. Set $V_{obs} = Obs(V, \Sigma_o)$

STEP 8. If $\mathcal{L}(V_{obs}) = \mathcal{L}(G_{sys})$, then set $R_{oe} = (X_R, \Sigma_e, f_R, \Gamma_R, x_{0,R}) = V$ and return $(True, R_{oe})$, else return *False*

In STEP 1 of Algorithm 3.2, automaton G_{sys} , which models the observable event occurrence in the system, is computed being equal to the observer of automaton G with respect to Σ_o . In STEP 2, automaton G_{int} that models the Intruder's state estimates, is computed as a copy of G_{sys} but renaming the events with subscript s .

STEP 3 computes automaton D with the current configuration of SD as proposed in [29]. Note that automaton D models every possible change of event order with respect to each channel delay. In STEP 4, automaton V is computed as the parallel composition of automata G_{sys} , D and G_{int} . Given that D has all the possible event occurrences and all changes in the order of its observations with respect to the Opacity-Enforcer property, the first intersection will synchronize only sequences such that its order of occurrence belong to $\mathcal{L}(G_{sys})$ and the second parallel composition erases the sequences whose changes in the order of event observation do not belong to $\mathcal{L}(G_{int})$, thereafter only sequences s_D such that $P_{e,o}(s_D) \in \mathcal{L}(G_{sys})$ and $P_{e,s}(s_D) \in \mathcal{L}(G_{int})$ will remain. It is important to note that the states of V , written as (x_{sys}, x_D, x_{int}) , and of the automata constructed from V contain, respectively, information about the current state in the plant, the events that are held by the Opacity-Enforcer and Intruder's state estimate.

STEP 5 removes from V the undesirable behaviors caused by the shuffling of event occurrences and observations by removing state $x_{rem} = (x_{sys}, x_D, x_{int})$ of automaton V , that satisfies at least one of the following conditions: (i) its third element, correspondent states in automaton G_{int} , which can also be interpreted as the state estimated by the intruder, belongs to the set of secret states of automaton G_{obs} , *i.e.*, $x_{int} \in X_{s,obs}$; (ii) its active event set is empty while its second element (events that have yet to be observed) is different from ν , *i.e.*, $\Gamma_V((x_{sys}, x_D, x_{int})) = \emptyset$ and $x_D \neq \nu$, where $x_{0,D} = \nu$ and ν represents “no event to be observed”; (iii) its active event set is different from that of its first element (correspondent state in automaton G_{sys}), *i.e.*, $(\Gamma_D(x_D) \cap \Gamma_{sys}(x_{sys})) \setminus \Gamma_V((x_{sys}, x_D, x_{int})) \neq \emptyset$, meaning that any event $\sigma \in \Sigma_o$ that could occur in x_{sys} was wrongly inhibited in x_{rem} even though further delays of events were allowed in automaton D . These steps intend to: (i) prevent a genuine or misleading estimation of secret states set, (ii) prevent the existence of states that have some events yet to be observed but with no successors (further possible events success of observation in its second element would not be inside the language generated by automaton G_{int} or would lead to already removed states) and

(iii) prevent the existence of states which some event occurrence in Σ_o that has been inhibited by previously removed states. Note that STEP 5 is repeated until no state is deleted and the accessible part operation is done whenever a state is removed.

The strategy of shuffling event occurrences with their observations is likely to leave V with decision conflicts, denoting that a state has events of Σ_o and Σ_s simultaneously in its active event set. When that occurs, the Opacity-Enforcer must either wait for the arrival of an event occurrence or release an event observation. These decision conflicts are removed in STEP 6, since for every state x_V of V , that has some observation event $\sigma_s \in \Sigma_s$ in its active event set, all transitions associated with events $\sigma \notin \Sigma_s$ departing from it will be deleted and in the sequel the accessible part operation over V is done. The idea behind this “cleaning” step is associated with the fact that the Opacity-Enforcer must release an observation whenever it is possible. In STEP 7 the automaton V_{obs} is computed as the observer of V with respect to Σ_o in order for the language generated by automaton V_{obs} to be compared with the language generated by automaton G_{sys} in STEP 8. This language comparison ensures that the Opacity-Enforcer model, given by automaton V , does not constrain the behavior of the system. If $\mathcal{L}(V_{obs}) = \mathcal{L}(G_{sys})$, then STEP 8 computes the realization of Opacity-Enforcer automaton $R_{oe} = V$. Note that automaton R_{oe} records every event occurrence and release event observations so as that the Intruder is not able to estimate any secret state of G_{obs} . In addition, STEP 8 returns $(True, R_{oe})$, meaning that the opacity-enforcement can be done with the current SD . On the other hand, if $\mathcal{L}(V_{obs}) \neq \mathcal{L}(G_{sys})$, then STEP 8 returns $False$, showing that the current SD does not allow the system to be CSO enforced.

We now will prove Algorithm 3.2 correctness. We start with the following result.

Lemma 3.1 *The projection over the set of observable events of the language generated by automaton V , built in STEP 4 of Algorithm 3.2, is the same as the language generated by automaton G_{obs} , i.e., $P_{e,o}(L(V)) = L_{obs}$.*

Proof: Given $V = G_{sys} || D || G_{int}$, let us deal with states $x_V = (x_{sys}, x_D, x_{int}) \in X_V$ in automaton D . We know that the state transitions in automaton G_{int} are

driven by events $\sigma_s \in \Sigma_s$, thus, no matter which component x_{sys} the parallel composition is in, all events $\sigma \in \Sigma_o$ are eligible to occur in x_V . With respect to the states of automaton D , we can say that there will exist two types of state: states where all events $\sigma \in \Sigma_o$ are eligible to occur; and states where no occurrence events are defined, thus, only events $\sigma_s \in \Sigma_s$ may occur in this type of state. Hence, no sequence of G_{sys} will be harmed while synchronizing it with a sequence of D , since if the component x_D represents a state of the former case, all events $\sigma \in \Sigma_o$ are eligible to occur, and; if it represents one of the latter case, only events $\sigma_s \in \Sigma_s$ are able to occur and will not harm any sequence of G_{sys} with respect to the occurrence behavior. Summarizing the presented facts, in a state $x_V = (x_{sys}, x_D, x_{int}) \in X_V$, either all events $\sigma \in \Sigma_o$ defined in x_{sys} are able to occur, since they are eligible in both components, x_D and x_{int} , or only event $\sigma_s \in \Sigma_s$ are eligible to occur, however, their occurrence will eventually lead V to a state where its components fit the former case, *i.e.*, all σ defined in x_{sys} may occur. Thus, while constructing V , the sequences of G_{sys} were extended but not harmed, hence, we have that $P_{e,o}(L(V)) = L(G_{obs})$. ■

Remark 3.1 Given $P_{e,s} : \Sigma_e^* \rightarrow \Sigma_{o,s}^*$, the opacity enforcement function OE can be extracted from the Opacity-Enforcer automaton $R_{oe} = (X_R, \Sigma_e, f_R, \Gamma_R, x_{0,R})$, which realizes $OE(s_e)$, as follows: $OE(s_e) = P_{e,s}(\Gamma_R(f_R(x_{o,R}, s_e)))$.

We now propose the following necessary and sufficient condition for CSO enforcement by changes in the order of event observations.

Theorem 3.2 The system modeled by automaton $G = (X, \Sigma, f, \Gamma, x_0)$ is CSO enforceable through changes in the order of event observations with respect to a observable event set $\Sigma_o \subseteq \Sigma$, a set of secret states $X_s \subseteq X$ and a Opacity-Enforcer bound $SD = \{(\sigma_1, k_1), (\sigma_2, k_2), \dots, (\sigma_n, k_n)\}$ if and only if the language generated by the verifier V_{obs} and language generated by the automaton G_{sys} are the same, *i.e.*, $\mathcal{L}(V_{obs}) = \mathcal{L}(G_{sys})$, as in Algorithm 3.2.

Proof: Let us consider $P_{e,o} : \Sigma_e^* \rightarrow \Sigma_o^*$ and $P_{e,s} : \Sigma_e^* \rightarrow \Sigma_{o,s}^*$.

(\Rightarrow) If $\mathcal{L}(V_{obs}) \neq \mathcal{L}(G_{sys}) \Rightarrow P_{e,o}(L(V)) \neq L_{obs}$, since $\mathcal{L}(V_{obs}) = L(O_{bs}(V, \Sigma_o)) = P_{e,o}(L(V))$ and $\mathcal{L}(G_{sys}) = L(G_{obs}) = L_{obs}$. The sentence $P_{e,o}(L(V)) \neq L_{obs}$ means that $\exists st \in L_{obs}$ and $\nexists s_e \in L(V) : P_{e,o}(s_e) = st$, therefore, $\exists s \in L_{obs}, \exists t \in L_{obs}/s, \forall n \in \mathbb{Z}_+, \forall u \in L_{obs}/st : |u| = n, \nexists s_e \in L(V) : P_{e,o}(s_e) = st u$, since the language generated by automata is prefix-closed. If $f_{obs}(x_{0,obs}, s) \in X_{s,obs}$, then we have that $\exists s \in L_{obs} : f_{obs}(x_{0,obs}, s) \in X_{s,obs}, \exists t \in L_{obs}/s, \forall n \in \mathbb{Z}_+, \forall u \in L_{obs}/st : |u| = n, \nexists s_e \in L(V) : P_{e,o}(s_e) = st u$, which implies the negation of Definition 3.4, therefore, the system is not enforceable through changes in the order of event observations with respect to Σ_o, X_s and SD . The assumption of s not reaching a secret state, *i.e.*, $f_{obs}(x_{0,obs}, s) \in X_{obs} \setminus X_{s,obs}$, is an absurd, since, from Lemma 3.1, the observable language generated by both automata, V and G_{obs} , were originally identical and, if the sequence st had never passed through any secret state of the observer, it means that Algorithm 3.2 had never shuffled its event observations. The case where $f_{obs}(x_{0,obs}, st) \in X_{s,obs}$ is equivalent to $s' = st$ and $t' = \varepsilon$. Hence, the system is not enforceable through changes in the order of event observations with respect to Σ_o, X_s and SD .

(\Leftarrow) If the language generated by the verifier V_{obs} is the same as the language generated by the automaton G_{sys} , which models the system, then, we have that:

$$L_{obs} = L(G_{sys}) = L(V_{obs}) = L(O_{bs}(V, \Sigma_o)) = P_{e,o}(L(V)) \quad (3.3)$$

Since automaton D generates sequences which are composed of the shuffling between event occurrences and observations, we have that $\forall s_e \in L(D), R^{-1}(P_{e,s}(s_e)) \in Pre(S_p(P_{e,o}(s_e)))$, and so, since automaton V is a subautomaton of D , $\forall s_e \in L(V), R^{-1}(P_{e,s}(s_e)) \in Pre(S_p(P_{e,o}(s_e)))$. Note that $L(V) \subseteq L(G_{sys} || D || G_{int}) \Rightarrow \forall s_e \in L(V), (P_{e,o}(s_e) \in L(G_{sys})) \wedge (P_{e,s}(s_e) \in L(G_{int})) \Rightarrow P_{e,o}(s_e), R^{-1}(P_{e,s}(s_e)) \in L(G_{sys}) \Rightarrow P_{e,o}(s_e), R^{-1}(P_{e,s}(s_e)) \in L_{obs}$.

The existence of some $s_e \in L(V) : \forall \sigma \in \Sigma_o, \mathcal{N}(s_e, \sigma) = \mathcal{N}(s_e, R(\sigma))$ is given by the fact that the delays are bounded, so eventually the second component of $x_V \in X_V$ will be ν and by condition (ii) of STEP 5 of Algorithm 3.2, where all

blocking states whose second component is not null were removed. Thus, $\forall s : f_{obs}(x_{0,obs}, s) \in X_{s,obs}, \forall t \in L_{obs}/s, \exists n \in \mathbb{Z}_+$ and $\exists u \in L_{obs}/st : |u| = n, \exists s_e \in L(V) : (P_{e,o}(s_e) = stu) \wedge (\mathcal{N}(s_e, \sigma) = \mathcal{N}(s_e, R(\sigma)), \forall \sigma \in \Sigma_o)$. Note that if $P_{e,o}(s_e) = stu$ and $\mathcal{N}(s_e, \sigma) = \mathcal{N}(s_e, R(\sigma)), \forall \sigma \in \Sigma_o$ holds, then it means that each event of stu had its observation released in a shuffled order by the Opacity-Enforcer, thus, $R^{-1}(P_{e,s}(s_e)) \in S_p(stu)$; the sentence $R^{-1}(P_{e,s}(s_e)) \in L_{obs}$ also holds, since $P_{e,s}(s_e) \in L(G_{int}) \Rightarrow P_{e,s}(s_e) \in R(L(G_{sys})) = R(L_{obs})$, therefore, $R^{-1}(P_{e,s}(s_e)) \in S_p(stu) \cap L_{obs}$. We can also affirm that $\forall s_e \in L(V), f_{obs}(x_{0,obs}, R^{-1}(P_{e,s}(s_e))) \notin 2^{X_s} \Rightarrow \forall s_e \in L(V), f_{obs}(x_{0,obs}, R^{-1}(P_{e,s}(s_e))) \in X_{obs} \setminus X_{s,obs}$, due to (i) condition in STEP 5. Thus $\forall s : f_{obs}(x_{0,obs}, s) \in X_{s,obs}, \forall t \in L_{obs}/s, \exists n \in \mathbb{Z}_+$ and $\exists u \in L_{obs}/st : |u| = n, \exists s_e \in L(V) : (R^{-1}(P_{e,s}(s_e)) \in S_p(stu) \cap L_{obs}) \wedge (\forall w \in Pre(R^{-1}(P_{e,s}(s_e))), f_{obs}(x_{0,obs}, w) \in X_{obs} \setminus X_{s,obs})$. Now, by setting $v = R^{-1}(P_{e,s}(s_e))$, we can write the statement as $\forall s : f_{obs}(x_{0,obs}, s) \in X_{s,obs}, \forall t \in L_{obs}/s, \exists n \in \mathbb{Z}_+, \exists u \in L_{obs}/st : |u| = n, \exists v \in S_p(stu) \cap L_{obs} : f_{obs}(x_{0,obs}, w) \in X_{obs} \setminus X_{s,obs}, \forall w \in Pre(v)$, which implies that, from Definition 3.4, the system is enforceable through changes in the order of event observations with respect to Σ_o, X_s and SD . ■

According to theorem 3.2, we can enforce CSO through changes in the order of event observations with respect to Σ_o, X_s and SD if and only if Algorithm 3.2 returns *True*.

3.3.3 An Algorithm for Finding Minimal Delay Bounds

Finding a feasible delay bound SD for the system to be CSO enforced is a hard task, since either the delay bounds will be not enough to enforce CSO or they will be unnecessarily large. In order to ease the search for a minimal feasible SD , the following algorithm finds, given upper bounds on the delays for each event. The algorithm returns *Failure*, if it is not possible to enforce-opacity with the current constraints.

Algorithm 3.3 *Optimization of step delays*

Input: $G, \Sigma_o, X_s, k^{max} = [k_1^{max}, \dots, k_n^{max}]$.

Output: $SD = \{(\sigma_1, k_1^{min}), (\sigma_2, k_2^{min}), \dots, (\sigma_n, k_n^{min})\}$ or *Failure*.

STEP 1. Construct $G_{obs} = O_{bs}(G, \Sigma_o)$.

STEP 2. Create a list $k = [k_1, \dots, k_n]$

STEP 3. Set $k_i = 0$ for $i = 1, \dots, n$

STEP 4. Set $SD = \{(\sigma_1, k_1), (\sigma_2, k_2), \dots, (\sigma_n, k_n)\}$

STEP 5. If $X_{obs} \cap 2^{X_s} = \emptyset$, then return SD , else:

STEP 5.1. While Algorithm 3.2 returns *False* and $k_i \neq k_i^{max}$, $i = 1, \dots, n$, do:

STEP 5.1.1. For $i = 1, 2, \dots, n$:

STEP 5.1.1.1. If $k_i < k_i^{max}$, then set $SD = \{(\sigma_1, k_1), \dots, (\sigma_i, k_i + 1), \dots, (\sigma_n, k_n)\}$

STEP 6. If Algorithm 3.2 returns *True*, then:

STEP 6.1. For $i = 1, 2, \dots, n$:

STEP 6.1.1. While Algorithm 3.2 returns *True* and $k_i \geq 0$, set $SD = \{\dots, (\sigma_i, k_i - 1), \dots\}$

STEP 6.1.2. Set $SD = \{\dots, (\sigma_i, k_i + 1), \dots\}$

Else, return *Failure*, opacity cannot be enforced with $SD = \{(\sigma_1, k_1^{max}), (\sigma_2, k_2^{max}), \dots, (\sigma_n, k_n^{max})\}$

STEP 7. Return SD

In STEP 1 of Algorithm 3.3, we compute the observer G_{obs} with respect to automaton G and Σ_o . In STEP 2 we create a list of delay $k = [k_1, k_2, \dots, k_n]$ and set all of its values to 0 in STEP 3. In STEP 4 we set the bound delay SD by

merging the informations from Σ_o and from k . In STEP 5, we check if the system is not already opaque by verifying if $X_{obs} \cap 2^{X_s} = \emptyset$. If it is already opaque, then return $SD = \{(\sigma_1, 0), \dots, (\sigma_n, 0)\}$, which means that G is CSO. On the other hand, if G is not CSO, then *Algorithm 3.3* proceeds to a loop where it searches for a SD such that the system can be CSO enforced through changes in the order of event observations. That loop works as follows: we check if opacity can be enforced with the current step delay configuration SD running *Algorithm 3.2* and if the answer is negative, each k_i is increased by 1, keeping in mind that if $k_i = k_i^{max}$ it will not be increased. STEP 5.1 ends when either opacity-enforcement is feasible with a given SD or k has reached its upper bound $k_i = k_i^{max}, \forall i = 1, \dots, n$. STEP 6 tries to find a minimal solution by checking the minimum value of each k_i such that opacity remains enforceable. If each k_i has reached its maximum value, *i.e.*, $k = k^{max}$, and *Algorithm 3.2* did not return *True*, it means that the system cannot be CSO enforced, hence, *Algorithm 3.3* returns *False* and implies that the current maximum delays are not enough to enforce CSO. If *Algorithm 3.2* has returned *True*, then STEP 6 starts with k_1 and decreases its value by 1 unit until either CSO can no longer be enforced (*Algorithm 3.2* returns *False*) or k_1 has reached a negative value; in the latter case, it sets that k_1 to the previous value by adding 1 to it. Then STEP 6 carries out this same procedure for elements k_2, k_3, \dots, k_n . Note that in STEP 6 we chose k_i in the ascending order to find a minimum solution but other solutions can be found depending on the order that k_i are chosen. Finally, STEP 7 returns a minimum step delay configuration SD such that G is CSO enforced by changing the order of event observation.

Remark 3.2 *Note that Algorithm 3.3 outputs one of many possible minimum solutions for the delays $[k_1, k_2, \dots, k_n]$. A way of obtaining the minimal solution is to find all the minimum solutions and use some criteria to find which ones among them are minimal, e.g. weights might be associated with each event delay and then the configuration that has the least cumulated weight will be the minimal solution.*

3.4 Example

To illustrate the opacity enforcing methodology proposed in the work, we use a classic problem called mouse in a maze [31], where the maze is composed of several rooms linked by one-way doors, which may or may not have sensors that fire when something passes through it. In the original problem, besides the mouse, there is also a cat, with both able to move freely throughout the rooms; supervisory control theory is then used in order for the mouse to always avoid the cat by locking or unlocking the doors, because if they meet, the cat will eat the mouse. Here, we replace the cat with traps placed beforehand in some rooms, which can be remotely activated by the Intruder. Both, the owner of the maze and the Intruder, know the rooms where the traps have been placed, and will be named as “Secret Rooms”. Our objective is, therefore, to prevent the Intruder from correctly estimating the mouse position with respect to the secret rooms.

The maze and its sensors are depicted in Figure 3.4, and the automaton G that models this system is shown in Figure 3.5. The room where the mouse is initially placed will be the initial state, *i.e.*, $X_0 = \{0\}$, and the set of secret states is $X_s = \{3, 5\}$. The set of events is partitioned as $\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$, where $\Sigma_o = \{a, b, c, r\}$ and $\Sigma_{uo} = \{\sigma_{uo}\}$ represent the signals issued by the sensors attached to some of the doors (note that different sensors may issue the same signal) and the passage through doors with no sensor at all, respectively. Since all doors are one-way and the mouse can get stuck inside some rooms, two one-way tubes are placed outside the maze to connect rooms 6 and 8 to the initial room 0. Note that, without the actions taken by the Opacity-Enforcer, both, Receiver and Intruder, would estimate the states as shown by the observer of G in Figure 3.6, and so, the Intruder can be sure that the mouse has reached a secret state when it estimates states $\{3\}$ or $\{5\}$.

We will now address the problem of designing an Opacity-Enforcer so as to achieve current-state opacity. To this end, we set $SD = \{(a, 1), (b, 1), (c, 0), (r, 0)\}$, *i.e.*, we will try to find a CSO enforcement policy that holds events a and b for at most one step after their occurrences and releases events c and r immediately after

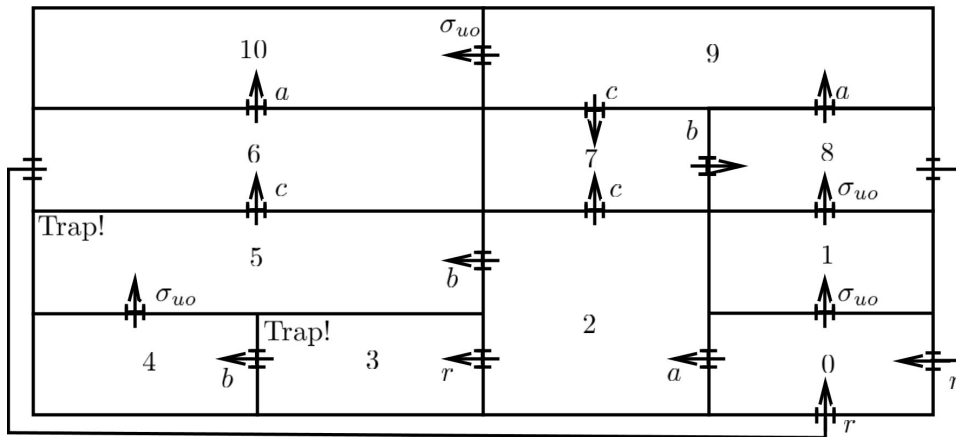


Figure 3.4: Mouse in a Maze problem structure.

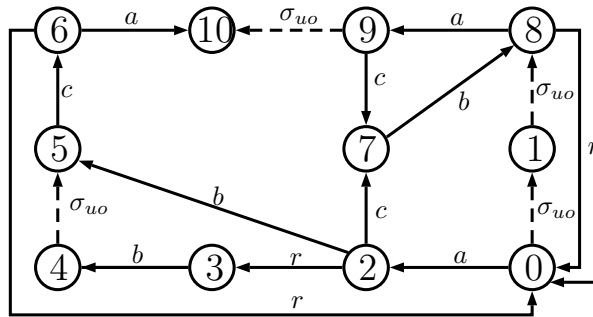


Figure 3.5: Automaton G that models the system dynamics.

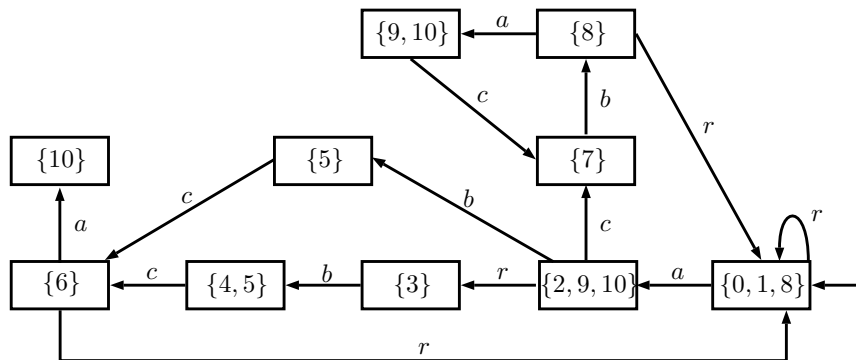


Figure 3.6: Observer automaton $G_{obs} = Obs(G, \Sigma_o)$.

their occurrences. We will start with *Algorithm 3.2*, used to verify if an Opacity-Enforcer with such bounds can be designed, as follows.

Example 3.6 *Let us consider the system shown in Figure 3.4, which is modeled by an automaton whose state transition diagram is depicted in Figure 3.5, where $X_0 = \{0\}$, $X_s = \{3, 5\}$, $\Sigma_o = \{a, b, c, r\}$, $\Sigma_{uo} = \{\sigma_{uo}\}$ and $\Sigma = \Sigma_o \cup \Sigma_{uo}$. In addition, the observer automaton of G is illustrated in Figure 3.6 and we set the Opacity-Enforcer bound property $SD = \{(a, 1), (b, 1), (c, 0), (r, 0)\}$. Starting *Algorithm 3.2*, we firstly build automaton G_{sys} in STEP 1, which is identical to G_{obs} . Following *Algorithm 3.2*, we then build automaton G_{int} , which is identical to G_{obs} , except for that subscript s has been added to all of its events, as stated in STEP 2. In the sequel, we construct automaton D with the Opacity-Enforcer bound property SD as in Example 3.5.*

*In the next step, we compute automaton $V = G_{sys} || D || G_{int}$ and then we proceed to STEP 5, where we delete undesirable states of V until there is no state that satisfies conditions (i), (ii) nor (iii). Figure 3.7 shows part of automaton V in which the painted states satisfy removal condition (i). After being deleted, the gridded states will also be removed, since they will no longer be accessible and then STEP 5 is repeated. After the painted and gridded states are all removed, the diagonally hatched state $(\{3\}, r, \{5\})$ will satisfy removal condition (ii) and then they will be removed, as well, and, in the sequel, the accessible part operation will performed and STEP 5 will be executed again. Finally, the diagonally hatched state $(\{2, 9, 10\}, \nu, \{2, 9, 10\})$ will be removed because it satisfies removal condition (iii). After that, no states satisfies the removal conditions, and thus, *Algorithm 3.2* proceeds to the next steps.*

In STEP 6, we remove from automaton V decision conflicts between waiting the arrival of an event occurrence or releasing an event observation, since, according to the problem formulation, the Opacity-Enforcer must release an event immediately after its observation neither harms the state estimation nor leads to a secret state estimate by the Intruder. Figure 3.8 shows the painted states $(\{8\}, b, \{7\})$ and $(\{9, 10\}, a, \{8\})$ in automaton V with decision conflicts, e.g, when the Opacity-

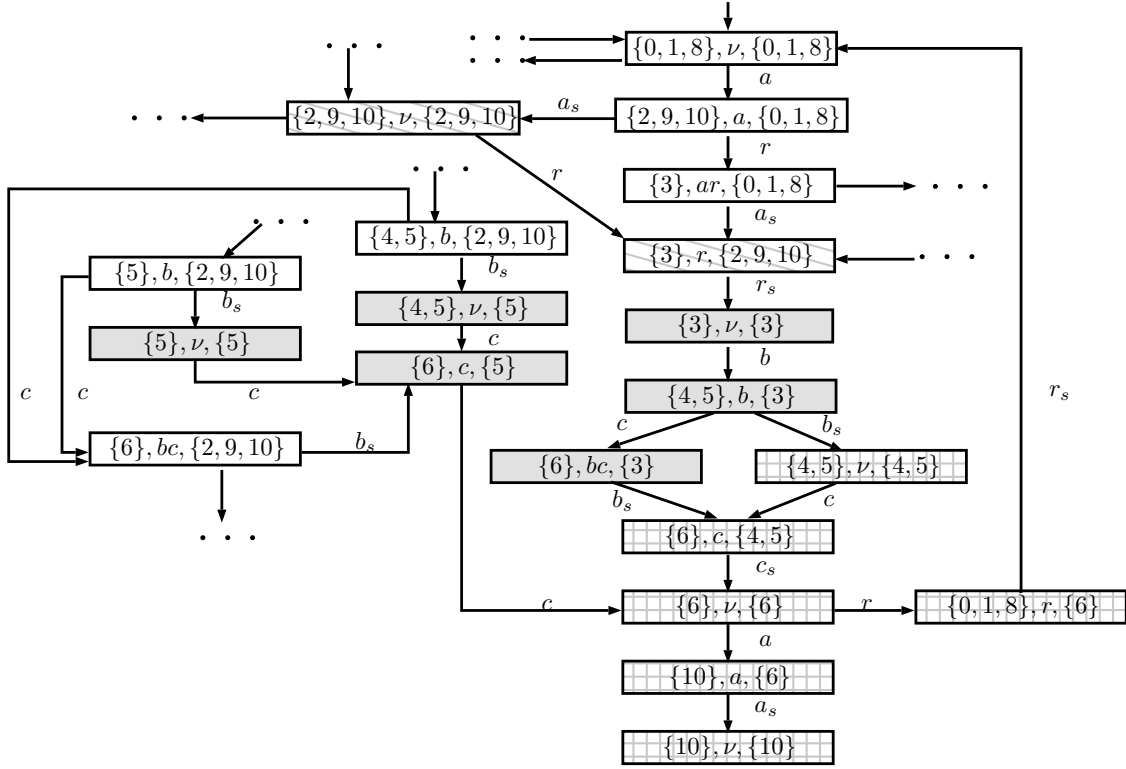
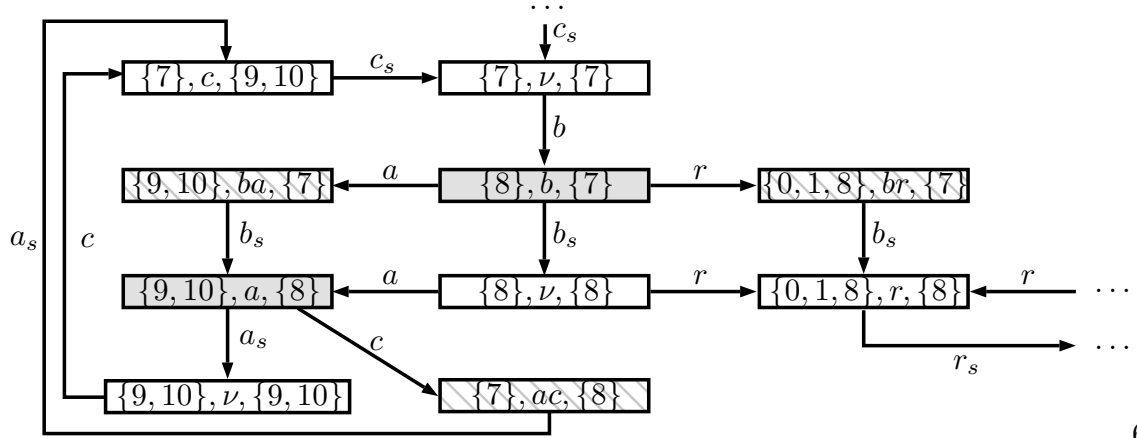


Figure 3.7: Part of V that shows undesirable states.

Enforcer is in state $(\{8\}, b, \{7\})$, it can either wait for events a or r to occur, or it can release the observation b_s . These decision conflicts are erased in STEP 6.1 when all transitions that does not have an event $\sigma_s \in \Sigma_{o,s}$ are removed from states $(\{8\}, b, \{7\})$ and $(\{9, 10\}, a, \{8\})$. As a consequence of those transitions removal, the hatched states will no longer be accessible and then they will be deleted.

The computation of the observer V_{obs} of automaton V with respect to Σ_o , $V_{obs} = O_{bs}(V, \Sigma_o)$ is done in STEP 7 and the comparison whether $\mathcal{L}(V_{obs}) = \mathcal{L}(G_{sys})$ is performed in STEP 8. As soon as the observer V_{obs} of the automaton V is done, it can be seen that both V_{obs} and G_{sys} generate the same language, thus, Algorithm 3.2 returns *True* about the system G being opacity-enforceable through changes in the order of event observations with respect to its observable event set $\Sigma_o = \{a, b, c, r\}$ and its step delays bound $SD = \{(a, 1), (b, 1), (c, 0), (r, 0)\}$ and also returns the realization of the Opacity-Enforcer automaton R_{oe} as shown in Figure 3.9, given by automaton V after removing the decision conflicts.



6

Figure 3.8: Part of V that shows decision conflicts.

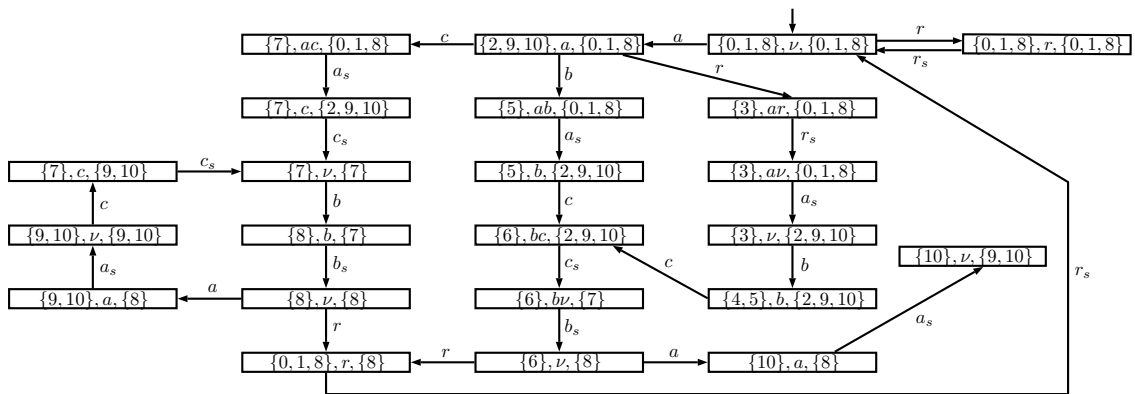


Figure 3.9: Automaton R_{oe} that realizes the opacity enforcement strategy.

It is important to emphasize the importance of language inclusion comparison between automaton V_{obs} and G_{sys} because, as shown throughout Example 3.6, Algorithm 3.2 shuffles all the possibilities of events occurrences and observations and then prunes all the unrealistic or undesirable possible changes in the order of event observation with respect to the bound of the Opacity-Enforcer. Note that if the upper bound of the Opacity-Enforcer is not enough to enforce opacity, eventually it will not release an event signal (every possible changes in the order of event observation would not guarantee opacity). In addition, these branches will be pruned, since holding an event forever is not plausible, therefore $\mathcal{L}(V_{obs}) \neq \mathcal{L}(G_{sys})$.

Example 3.7 *Regarding Algorithm 3.3, let us input the same G , Σ_o , X_s used in Example 3.6 and set $k^{max} = [2, 2, 2, 0]$, then Algorithm 3.3 will build the observer of automaton G , shown in Figure 3.6, in STEP 1, create a list k and set $k = [0, 0, 0, 0]$ in STEP 2 and STEP 3, respectively. Then, it will set $SD = \{(a, k_1), (b, k_2), (c, k_3), (r, k_4)\} = \{(a, 0), (b, 0), (c, 0), (r, 0)\}$ in STEP 4. In STEP 5, since $X_{obs} \cap 2^{X_s} = \{\{3\}, \{5\}\}$, we proceed to STEP 5.1, where we will run Algorithm 3.2 and it will return False, thus increasing the step delays to $SD = \{(a, 1), (b, 1), (c, 1), (r, 0)\}$. Note that k_4 has not increased because it has already reached its maximum value $k_4^{max} = 0$. We repeat STEP 5 with the updated SD Algorithm 3.2 returns True, allowing Algorithm 3.3 to proceed to STEP 6, where it tries to minimize k . Firstly, STEP 6.1.1 targets k_1 and Algorithm 3.2 returns True, then we decrease k_1 setting $SD = \{(a, 0), (b, 1), (c, 1), (r, 0)\}$. The second time we run STEP 6.1.1, it still targets k_1 but Algorithm 3.2 returns False, then it goes to STEP 6.1.2, where we set $SD = \{(a, 1), (b, 1), (c, 1), (r, 0)\}$ back and then STEP 6.1 change its target to k_2 , repeating the whole process. Finally, Algorithm 3.3 will return $SD = \{(a, 1), (b, 1), (c, 0), (r, 0)\}$ in STEP 7.*

Remark 3.3 *Note that automaton R_{oe} , shown in Figure 3.9, keeps track of both the system dynamics and the observation released to the Intruder. For example, if G generates the sequence $s = rab$, then the mouse will be in the secret room 5, but the Opacity-Enforcer releases events r_s and a_s , holds b , then waits for event c to*

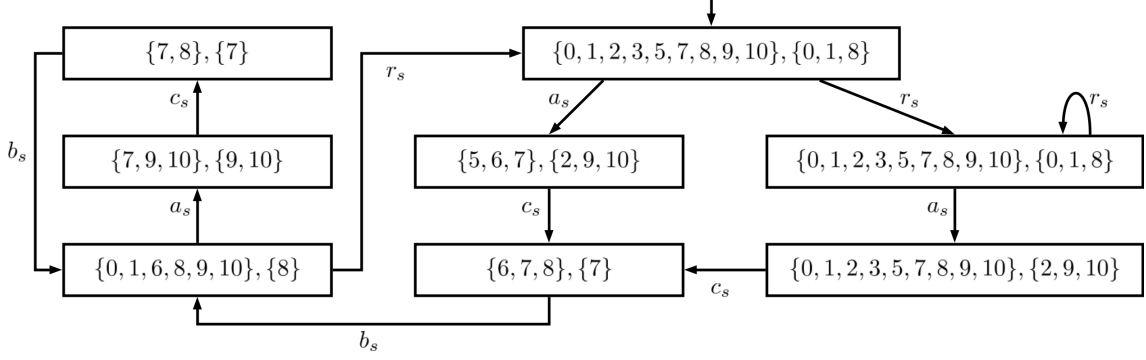


Figure 3.10: Estimator Automaton \mathcal{E} .

occur in order to release $c_s b_s$, generating $r_s a_s c_s b_s$; thus the Intruder estimates the mouse current position as in rooms 2, 9 or 10 while the mouse is actually in room 5. Note that, since the receiver also has the knowledge on the OE dynamics, it will have access to the current state of R_{oe} , i.e., $x_{oe} = \{\{5\}, b, \{2, 9, 10\}\}$, meaning that for the receiver, the mouse is in room 5. This is a significant improvement over existing opacity-enforcement methodologies, and suggests that there may exist an inverter automaton that is capable to make possible the determination of the exact state of the system by the receiver.

Notice that the automaton which models both, the estimation of an outsider which knows about the opacity enforcer function and the estimation of the intruder, is computed by building the observer automaton $\mathcal{E} = O_{bs}(R_{oe}, \Sigma_{o,s})$ and renaming its states according to each state estimation component. The following example shows the Estimator Automaton $\mathcal{E} = (X_{\mathcal{E}}, \Sigma_{o,s}, f_{\mathcal{E}}, \Gamma_{\mathcal{E}}, x_{0,\mathcal{E}})$ of the system presented in Example 3.6.

Example 3.8 Let us consider the system presented in Example 3.6 and its Opacity-Enforcer depicted in Figure 3.9. In order to build the estimator automaton \mathcal{E} , after we compute the observer of R_{oe} , for each state of $O_{bs}(R_{oe}, \Sigma_{o,s})$, we: (i) merge all x_{sys} of $x_{oe} = \{x_{sys}, x_D, x_{int}\}$ in the first element of $x_{\mathcal{E}}$; (ii) discard the second element x_D , and; (iii) take x_{int} as the second element of $x_{\mathcal{E}}$. For example, the initial state of $O_{bs}(R_{oe}, \Sigma_{o,s})$ is composed of states $(\{0, 1, 8\}, \nu, \{0, 1, 8\})$, $(\{0, 1, 8\}, r, \{0, 1, 8\})$, $(\{2, 9, 10\}, a, \{0, 1, 8\})$, $(\{5\}, ab, \{0, 1, 8\})$, $(\{7\}, ac, \{0, 1, 8\})$ and $(\{3\}, ar, \{0, 1, 8\})$,

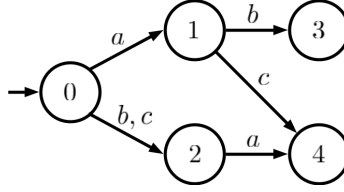


Figure 3.11: Automaton G .

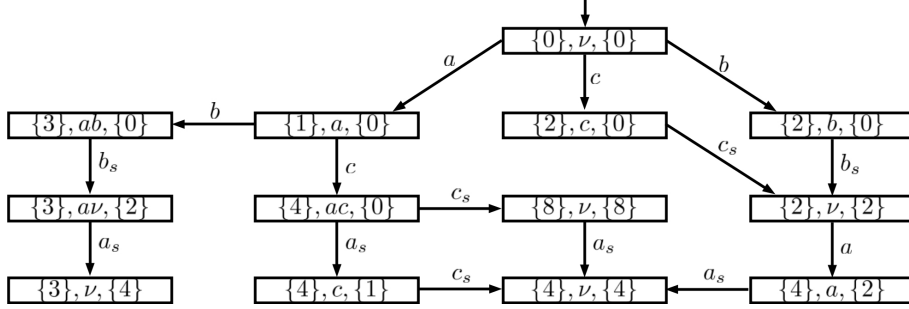


Figure 3.12: Opacity-Enforcer automaton of G .

thus, we set $x_{0,\mathcal{E}} = (\{0, 1, 8\} \cup \{2, 9, 10\} \cup \{5\} \cup \{7\} \cup \{3\}, \{0, 1, 8\}) = (\{0, 1, 2, 3, 5, 7, 8, 9, 10\}, \{0, 1, 8\})$.

Figure 3.10 shows the state transition graph of automaton \mathcal{E} . Notice that, even though the estimator automaton \mathcal{E} has states whose elements represent the same estimation, they cannot be merged, since their future behaviors are different.

The following example shows that the proposed algorithms cannot deal with decision conflicts caused by $|OE(s_e)| > 1$.

Example 3.9 Let a system be modeled by the automaton G depicted in Figure 3.11, and let $\Sigma = \Sigma_o = \{a, b, c\}$, $X_s = \{3\}$ and $SD = [(a, 1), (b, 0), (c, 0)]$. Since $\Sigma = \Sigma_o$, G_{obs} is also represented by the state transition diagram shown in Figure 3.11. The Opacity-Enforcer automaton of G , synthesized by the proposed algorithms, is shown in Figure 3.12. Note that both observation releases, a_s and c_s , are allowed in state $(\{4\}, ac, \{0\})$, reached by the sequence $s_e = ac$, i.e., $OE(ac) = \{a_s, c_s\}$.

Remark 3.4 The opacity enforcement strategy proposed throughout this work can also be applied with a view to enforce LBO, ISO or IFO, since these notions can be transformed to CSO, as presented by WU and LAFORTUNE [8].

Chapter 4

Conclusion and future works

We have presented in this work a new methodology for enforcing CSO of DES modeled by automaton that leverages the possibility of delaying event occurrence observation to make changes in its order of observation, misleading the intruder to wrongly estimate that have either both secret and non-secret components or non-secret components only.

The Opacity-Enforcer proposed in this work keeps track not only of the events executed by the system but also of the release of their observation signals. Such a feature can be regarded as an advance in opacity-enforcement strategies, since it suggests that both opacity, from the Intruder's point of view, and correct estimation, from the Opacity-Enforcer's point of view, may be achieved. As far as the authors know, no previous works had dealt with the problem of achieving both opacity and information transmission.

Future Works

The methodology proposed here can be enhanced by also allowing event erasure, which could be done by applying the dilation operation proposed in [32] to automaton D , where the intermittent loss of event observation will be seen as the event eligibility to erasure.

Further researches can also explore this opacity-enforcement methodology in the coordinated architecture context, where there are multiples channels and possibly

multiple Intruders trying to discover the secrets of the system.

In addition, the possibility of enforcing opacity on a system with secrets in the perspective of the Intruder but keeping the estimation of some “goal states” not dubious with respect to the receivers can also be explored as an extension to this work.

Bibliography

- [1] BRYANS, J. W., KOUTNY, M., RYAN, P. Y. “Modelling Opacity Using Petri Nets”, *Electronic Notes in Theoretical Computer Science*, v. 121, n. Supplement C, pp. 101–115, 2005. Proceedings of the 2nd International Workshop on Security Issues with Petri Nets and other Computational Models (WISP 2004).
- [2] JACOB, R., LESAGE, J.-J., FAURE, J.-M. “Overview of discrete event systems opacity: Models, validation, and quantification”, *Annual Reviews in Control*, v. 41, pp. 135–146, 2016.
- [3] LIN, F. “Opacity of discrete event systems and its applications”, *Automatica*, v. 47, pp. 496–503, 2011.
- [4] BADOUEL, E., BEDNARCZYK, M., BORZYSZKOWSKI, A., CAILLAUD, B., DARONDEAU, P. “Concurrent secrets”, *Discrete Event Dynamic Systems: Theory and Applications*, v. 17, n. 4, pp. 425–446, 2007.
- [5] DUBREIL, J., DARONDEAU, P., MARCHAND, H. “Supervisory control for opacity”, *IEEE Transactions on Automatic Control*, v. 55, n. 5, pp. 1089–1100, 2010.
- [6] SABOORI, A., HADJICOSTIS, C. N. “Notions of security and opacity in discrete event systems”. In: *46th IEEE Conference on Decision and Control*, pp. 5056–5061, 2007.
- [7] SABOORI, A., HADJICOSTIS, C. N. “Verification of initial-state opacity in security applications of DES”. In: *9th International Workshop on Discrete Event Systems (WODES)*, pp. 328–333, 2008.
- [8] WU, Y.-C., LAFORTUNE, S. “Comparative analysis of related notions of opacity in centralized and coordinated architectures”, *Discrete Event Dynamic Systems: Theory and Applications*, v. 23, n. 3, pp. 307–339, 2013.

- [9] SABOORI, A., HADJICOSTIS, C. N. “Verification of infinite-step opacity and analysis of its complexity”, *IFAC Proceedings Volumes*, v. 42, n. 5, pp. 46–51, 2009.
- [10] CASSEZ, F. “The Dark Side of Timed Opacity”. In: *Advances in Information Security and Assurance: Third International Conference and Workshop, ISA 2009, Seoul, Korea, June 25-27, 2009. Proceedings*, pp. 21–30, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [11] SABOORI, A., HADJICOSTIS, C. N. “Coverage analysis of mobile agent trajectory via state-based opacity formulations”, *Control Engineering Practice*, v. 19, pp. 967–977, 2011.
- [12] DUBREIL, J., JÉRON, T., MARCHAND, H. “Monitoring Confidentiality by Diagnosis Techniques”, *Proceedings of the European Control Conference*, pp. 2584–2589, 2009.
- [13] SABOORI, A., HADJICOSTIS, C. N. “Verification of K -step opacity and analysis of its complexity”, *IEEE Transactions on Automation Science and Engineering*, v. 8, n. 3, pp. 549–559, 2011.
- [14] YIN, X., LAFORTUNE, S. “On two-way observer and its application to the verification of infinite-step and K -step opacity”. In: *13th International Workshop on Discrete Event Systems (WODES)*, pp. 361–366, 2016.
- [15] YIN, X., LAFORTUNE, S. “A new approach for the verification of infinite-step and k -step opacity using two-way observers”, *Automatica*, v. 80, pp. 162–171, 2017.
- [16] YIN, X., LAFORTUNE, S. “A new approach for synthesizing opacity-enforcing supervisors for partially-observed discrete-event systems”. In: *American Control Conference (ACC)*, pp. 377–383, 2015.
- [17] WU, Y.-C., LAFORTUNE, S. “Synthesis of insertion functions for enforcement of opacity security properties”, *Automatica*, v. 50, n. 5, pp. 1336–1348, 2014.
- [18] WU, Y.-C., LAFORTUNE, S. “Synthesis of opacity-enforcing insertion functions that can be publicly known”. In: *IEEE 54th Annual Conference on Decision and Control (CDC)*, pp. 3506–3513, 2015.
- [19] WU, Y.-C., LAFORTUNE, S. “Synthesis of optimal insertion functions for opacity enforcement”, *IEEE Transactions on Automatic Control*, v. 61, n. 3, pp. 571–584, 2016.

- [20] FALCONE, Y., MARCHAND, H. “Runtime enforcement of K-step opacity”. In: *IEEE 52nd Annual Conference on Decision and Control (CDC)*, pp. 7271–7278, 2013.
- [21] FALCONE, Y., MARCHAND, H. “Enforcement and validation (at runtime) of various notions of opacity”, *Discrete Event Dynamic Systems: Theory and Applications*, v. 25, n. 4, pp. 531–570, 2015.
- [22] SABOORI, A., HADJICOSTIS, C. N. “Opacity-enforcing supervisory strategies for secure discrete event systems”. In: *47th IEEE Conference on Decision and Control (CDC)*, pp. 889–894, 2008.
- [23] CASSEZ, F., DUBREIL, J., MARCHAND, H. “Synthesis of opaque systems with static and dynamic masks”, *Formal Methods in System Design*, v. 40, n. 1, pp. 88–115, 2012.
- [24] TONG, Y., LI, Z., SEATZU, C., GIUA, A. “Current-state opacity enforcement in discrete event systems under incomparable observations”, *Discrete Event Dynamic Systems: Theory and Applications*, pp. 1–22, 2017. doi: 10.1007/s10626-017-0264-7.
- [25] WU, Y.-C., RAMAN, V., RAWLINGS, B. C., LAFORTUNE, S., SESHIA, S. A. “Synthesis of Obfuscation Policies to Ensure Privacy and Utility”, *Journal of Automated Reasoning*, v. 60, n. 1, pp. 107–131, 2018.
- [26] JI, Y., LAFORTUNE, S. “Enforcing Opacity by Publicly Known Edit Functions”. In: *56th IEEE Conference on Decision and Control*, pp. 377–383, 2017.
- [27] KEROGLOU, C., LAFORTUNE, S. “Verification and Synthesis of Embedded Insertion Functions for Opacity Enforcement”. In: *56th IEEE Conference on Decision and Control*, pp. 377–383, 2017.
- [28] BEN-KALEFA, M., LIN, F. “Opaque superlanguages and sublanguages in discrete event systems”. In: *Proceedings of the 48th IEEE Conference on Decision and Control/28th Chinese Control Conference*, pp. 199–204, 2009.
- [29] NUNES, C. E. V., MOREIRA, M. V., ALVES, M. V. S., CARVALHO, L. K., BASILIO, J. C. “Codiagnosability of networked discrete event systems subject to communication delays and intermittent loss of observation”, *Discrete Event Dynamic Systems: Theory and Applications*, pp. 1–32, 2018. doi: 10.1007/s10626-017-0265-6.

- [30] CASSANDRAS, C. G., LAFORTUNE, S. *Introduction to Discrete Events Systems*. 2nd ed. New York, NY : USA, Springer, 2008.
- [31] RAMADGE, P. J., WONHAM, W. M. “The control of discrete event systems”, *Proceedings of the IEEE*, v. 77, n. 1, pp. 81–98, 1989.
- [32] CARVALHO, L. K., BASILIO, J. C., MOREIRA, M. V. “Robust diagnosis of discrete event systems against intermittent loss of observations”, *Automatica*, v. 48, pp. 2068–2078, 2012.