Verification and Enforcement of Opacity Security Properties in Discrete Event Systems

by

Yi-Chin Wu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering: Systems)
in The University of Michigan
2014

To Mom, Dad, Sister, who have given me their endless love and support,

and to Taiwan, the country that has nurtured me.

ACKNOWLEDGEMENTS

I am so grateful to have the friendships, support, and encouragement from so many people on the journey of my PhD.

This dissertation would not have been possible without the guidance of my research advisor Professor Stéphone Lafortune. He is the person who exposed me to Discrete Event Systems and helped me gain appreciation for it. He is a great academic advisor who is always patient and excited about the little progress I had achieved. He is also a life mentor who has taught me and my fellow students how to balance our academic work and personal lives, and presents us how to eagerly and humbly pursue the knowledge of, especially, but not limited to Discrete Event Systems.

I would also like to thank Professor Demosthenis Teneketzis for being an important mentor and teacher in my graduate life. He always makes efforts to advance my and my fellow students' intellectual growth. He spends time to get to know our personal backgrounds, and encourages us to pursue our own goals.

I thank my other thesis committee members: Professor Feng Lin, Professor Achilleas Anastasopoulos, Professor Kevin Compton, Professor Alfred Hero, and Professor Karem A. Sakallah. Their encouragement and comments throughout my PhD have helped me to become what I am.

I thank Professor Edward Lee at UC Berkeley and Professor Raymond Kwong at University of Toronto, Doctor Colleen Swanson, and Doctor Vasu Raman for their appreciation of my work and for generously sharing their criticisms and visions with me. They have given me new perspectives on the integration of Discrete Event Systems in security, which serve as a key source of encouragement for me to pursue further in this field.

I would like thank Hongwei Liao, Eric Dallal, Yin Xiang, Maria Paola Cabasino, Romulo Goes, and Antoine Butez for their discussion on Discrete Event systems and their companies in EECS 4400.

My PhD would not have been so enjoyable without the friendships of many EE:Systems peers. I still remember vividly when Patricia Laskowsky, Sammy Lee, Mike Allison, Takanori Watanabe, Bhargav Avasarala, Chun Lo, Sanghoon Kim, Sungjoon Park, Mads Almassalkhi, Hamid Ossareh and I studied EECS 501 together in our first year. Since then, we have built our friendships, and we have helped and encouraged the growth of one and another. I thank Joyce Liu, Parinaz Naghizadeh, Qingsi Wang, Arun Padakandla, Curtis Jin, Mai Le, and Yelin Kim for letting me stop by their offices for technical and non-technical chats. Also, my special thanks go to Becky Turanski, for being such a great “mother” of the EE:Systems kids. I also thank the *CRB volleyball friends for showing and teaching me how to play high quality games. Of course, I thank all my friends in Taiwan although we are thousands miles apart. Last, I thank all my roommates, and thank the Teatime friends, a group of Taiwanese friends who gather together on Friday nights, drinking tea and having intellectual discussions and jokes. Being together with them is the best way to cure homesickness.

Foremost, I thank my parents, Der-Lin Wu and Fong-Li Lu, for their constant support, love and for being my lifelong mentors; my sister Yi-Lin Wu, for being a cheerleader and for always providing her lovely humor. I thank my grandmom, Yu-Hua Chang Lu, for loving me in her whole life. I also want to thank Shang-Pin Sheng for always being there to share my achievements and struggles, and for being the best comrade in our PhD journeys.

Finally, I wish to acknowledge the financial support from NSF.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Verification and Enforcement of Opacity Security Properties
in Discrete Event Systems

by

Yi-Chin Wu

Chair: Stéphane Lafortune

The need for stringent cybersecurity is becoming significant as computers and networks are integrated into every aspect of our lives. A recent trend in cybersecurity research is to formalize security notions and develop theoretical foundations for designing secure systems. In this dissertation, we address a security notion called *opacity* based on the control theory for Discrete Event Systems (DES). Opacity is an information-flow property that captures whether a given secret of the system can be inferred by intruders who passively observe the behavior of the system. Finite-state automata are used to capture the dynamics of computer systems that need to be rendered opaque with respect to a given secret. Under the observation of the intruder, the secret of the system is opaque if "whenever the secret has occurred, there exists another non-secret behavior that is observationally equivalent."

This research focuses on the analysis and the enforcement of four notions of opacity. First, we develop algorithms for verifying opacity notions under the attack model of a single intruder and that of multiple colluding intruders. We then consider the enforcement of opacity when the secret is not opaque. Specifically, we propose a novel enforcement mechanism based on event insertion to address opacity enforcement for a class of systems whose dynamics cannot be modified. An insertion function, placed at the output of the system, inserts fictitious observable events to the system's output without interacting with the system. We develop a finite structure called the All-Insertion Structure (AIS) that enumerates all valid insertion functions. The AIS establishes a necessary and sufficient condition for the existence of a valid insertion function, and provides a structure to synthesize one insertion function. Furthermore, we introduce the maximum total cost and the maximum mean cost to quantify insertion functions. A condition for determining which cost objective to use is established. For each cost, we develop an algorithmic procedure for synthesizing an optimal insertion function from the AIS. Finally, our analysis and enforcement procedure is applied to ensuring location privacy in location-based services.

CHAPTER I

Introduction

Cybersecurity is an increasingly important issue as computers and networks are integrated into every aspect of our lives. Many efforts have been made to develop secure and reliable systems. However, existing methods often rely on the experience of software engineers and cannot be easily adapted to different application domains. We need a *science of cybersecurity*, which provides theoretical foundations for designing secure systems in application domains, and which can be used to predict attacks that exploit previously unknown vulnerabilities [36, 50].

One major goal in the development of a science of cybersecurity is to construct secure systems that are *verifiable*. In this dissertation, we address the problem of designing verifiable secure systems based on the control theory for Discrete Event Systems (DES). Specifically, central to this research is a security property called *opacity*, which is a general information-flow property that characterizes whether a given *secret* information can be inferred by an outside observer or not. Finite-State Automata (FSA) are used to describe the behavior of computer systems that need to be rendered opaque with respect to a given secret. The secret of the system is opaque if the intruder, which partially observes the system's behavior, is never sure whether the secret of the system has occurred or not. By leveraging techniques from DES, we formalize opacity notions, analyze opacity, design opacity enforcement strategies, and perform optimal enforcement.

1.1 Overview of Opacity Notions

The notion of opacity was first introduced in the computer science community to analyze whether the key used in the cryptographic protocol can be inferred by an outside observer [35]. It then became an active research topic in DES, as this class of dynamic systems provides suitable formal models and analytical techniques for investigating opacity. Specifically, in [8], opacity was investigated in systems modeled as Petri nets, where the secret is defined as predicates over Petri net markings (i.e., states). Following this work, the authors in [9] investigated opacity in labelled transition systems, where the secret was defined as predicates over runs. Similar work is the notion of *secrecy* defined in labelled transition systems [1]. Later, the works in [42, 43, 45] considered opacity notions using finite-state automata models and investigated different notions of state-based opacity. Recently, the study of opacity has also been extended to recursive tile systems [14] and pushdown systems [30].

In this dissertation, we focus on the notions of opacity in DES modeled as FSA. The ingredients of the FSA formulation of an opacity problem are: (1) the system has a *secret*; (2) the system is modeled as a partially observable and/or nondeterministic automaton; (3) the *intruder* is a passive observer that has full knowledge of the system structure. The secret of the system is *opaque* if "whenever the secret has occurred, there exists another *non-secret* behavior that is observationally equivalent." That is, the intruder is never sure if the secret has occurred. The system is guaranteed the "plausible deniability" of the secret.

The secret of the system can be defined using any representation in the given automaton model, such as states and languages. With different representations of the secret, various notions of opacity have been introduced in the literature, including but not limited to: (i) Initial-State Opacity where the secret is a set of initial states; (ii)

Current-State Opacity where the secret is a set of current states; (iii) Language-Based Opacity where the secret is a subset of the set of system runs. (iv) Initial-and-Final-State Opacity where the secret is a set of initial and finite states pairs. We will focus on these four notions of opacity in this dissertation.

Methods for verifying if the given secret is opaque or not have been investigated in [11, 33, 43, 59]. The work in [20] also develops techniques for detecting and predicting the flow of secret information based on the diagnosis technique in [48]. In this dissertation, we review existing verification methods and propose new verification method for opacity notions. When a secret is not opaque, the ensuing question is how to enforce the secret to be opaque. In this regard, many enforcement mechanisms have been proposed. In [19, 46, 56, 4, 5], the authors consider the design of an opacity-enforcing supervisory controller, which disables the behavior of the system when the secret is going to be revealed, based on the supervisory control theory of DES. The authors of [11] enforce opacity using dynamic observers, which dynamically modify the observability of every event by activating and deactivating sensors. Another enforcement approach that allows the full system behavior is the run-time enforcement mechanism in [23]. In that work, the authors employ delays at run-time when the system outputs executions in order to enforce K-step opacity. Also, the control problem of the so-called concurrent secrecy is solved in [3]. In contrast to these methods, we develop in this dissertation a novel opacity enforcement mechanism based on insertion functions. Such an insertion function remedies opacity by inserting additional observable events to the output of the system. This insertion mechanism is suitable for the four opacity notions considered in this dissertation.

Once an opacity enforcement mechanism is chosen, the next interesting question that arises is whether there is an opacity enforcer that is *optimal* in some specific sense. For the approach of opacity-enforcing supervisory controllers, optimality refers to restricting the system's behavior in a minimal manner [19, 46]. For the use of dynamic observers, the authors in [11] assign a cost value to the activation of the sensor of each event, and leverage game theoretical results to synthesize an optimal dynamic observer that consumes the least worst-case average cost. Inspired by the approach in [11], in this dissertation, we consider the synthesis of an optimal insertion function. Specifically, we assign a cost value to each inserted event and synthesize an optimal insertion function with respect to the total cost and the average cost (per system output), both in the worst case.

1.2 Organization and Main Contributions

The main contributions and the organization of this dissertation are summarized as follows.

Chapter II: Opacity Problems in Automata Formulations

This chapter reviews some basics in Discrete Event Systems and formally formulates four notions of opacity that we consider in this dissertation: Current-State Opacity (CSO), Initial-State Opacity (ISO), Language-Based Opacity (LBO), and a new notion called Initial-and-Final-State Opacity (IFO) that we introduce.

Chapter III: Verification of Opacity Notions in Centralized and Coordinated Architectures ([59])

In this chapter, we review existing methods for verifying CSO and LBO, leverage an existing algorithm for verifying ISO to verify IFO, and develop a more efficient algorithm that reduces the complexity for verifying ISO from $O(2^{|X|^2})$ to $O(2^{|X|})$. We establish polynomial reductions between all pairs of opacity notions. These reductions between notions allow us to verify one opacity notion by transforming it to another opacity notion, which provide a basis for developing opacity enforcement mechanisms that work for all four notions.

We also investigate opacity notions in the scenario of multiple colluding intruders. Specifically, we consider CSO, ISO, and IFO under a coordinated attack model where a coordinator aggregates all intruders' state estimates. For each notion, a characterization of the corresponding notion of *joint opacity* and an algorithmic procedure for its verification are provided.

Chapter IV: Opacity Enforcement Using Event Insertion ([58, 60])
This chapter considers the problem of enforcing opacity when the secret is not opaque. In view of the limitations of existing opacity enforcement methods, we propose a novel enforcement mechanism based on event insertion. Specifically, the approach based on supervisory control enforces opacity by disabling the system's behavior; thus, it does not apply to systems that must execute their full behaviors. The use of a dynamic observer allows the full system behavior; but it changes the observable behavior of the system, which may reveals clues about the defense model to the intruder. Also, using delays to enforce opacity only applies to secrets for which time duration is of concern.

An insertion function in our enforcement mechanism is a monitoring interface placed at the output of the system, as shown in Figure 1.1. It monitors the system's output behavior and inserts fictitious observable events to the output without interacting with the system. The intruder is assumed to have no knowledge of the

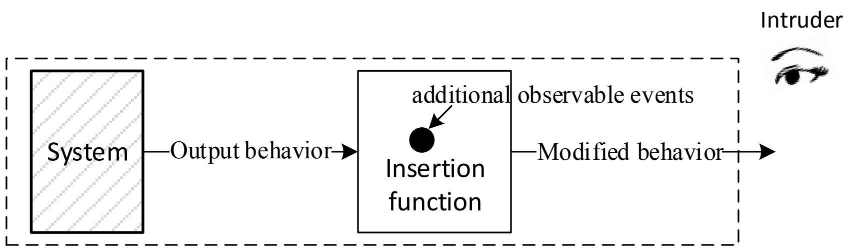


Figure 1.1: The insertion mechanism

insertion function at the outset. We want to ensure that the intruder never suspects the existence of an insertion function. Hence, fictitious events need to be inserted in a “convincing” manner – where the modified output is always consistent with an existing behavior that does not reveal the secret. We formulate the mathematical conditions for the interface requirement and the requirement for convincing insertions. We refer to insertion functions that satisfy both requirements as *i-enforcing*. Our goal is to synthesize one i-enforcing insertion function. We start with solving the problem of whether or not there exists an i-enforcing insertion function by constructing the “All Insertion Structure” (AIS). The AIS, as it is called, is a finite structure that enumerates *all* i-enforcing insertion functions. Specifically, the AIS is a game structure that enumerates all insertion function’s “winning strategies” that react to the output from the system. We use the AIS to provide a necessary and sufficient condition for the existence of an i-enforcing insertion function, and develop an algorithmic procedure that synthesizes an i-enforcing insertion function from the AIS.

Chapter V: Synthesis of Optimal Insertion Functions for Opacity Enforcement ([61])

This chapter considers the problem of synthesizing an *optimal* insertion function that minimizes the overhead cost introduced by insertion. We use the AIS as the structure to solve this optimal synthesis problem, as it enumerates all valid insertion functions. We assign to each inserted event a non-negative integer cost value. It turns out that an insertion function may need to insert an infinite number of events and thus incur an infinite total cost. With this consideration, we solve two optimization problems, one with respect to the *maximum total cost* and the other with respect to the *maximum mean cost*. The former captures the total insertion cost and the latter considers the average insertion cost (per system output), both in the worst-case scenario.

We first compute on the AIS the optimal *maximum total cost* for insertion functions. If this value is finite, we synthesize an optimal *total-cost* insertion function. Otherwise, we construct an optimal *mean-cost* insertion function. The synthesis of an optimal insertion function is formulated as finding an optimal insertion strategy on the AIS. We leverage the game theoretical results developed for *minmax games* or *mean payoff games*, depending on which objective cost function is used. In either case, an algorithmic procedure that encodes the optimal insertion strategy as a finite-state I/O automaton is provided.

Chapter VI: Case Study: Ensuring Privacy in Location-Based Services Using Opacity Techniques ([62])

This chapter considers the application of Location-Based Services (LBS) and use opacity techniques to solve the issue of location privacy. Nondeterministic automata are used to model the user's moving patterns. We use opacity techniques to prove that the existing method of *location anonymizers*, which blurs the user's accurate location to a region of locations, is insufficient to provide location privacy when the user continuously makes queries. To enforce location privacy, we propose to add to the anonymizer an *insertion function* that inserts fictitious queries. The design of such an insertion function is adapted from that for opacity, to fit the application of LBS. We also follow the optimization procedure in Chapter V to design an *optimal* insertion function that minimizes the overhead insertion cost. Overall, the insertion functions are added-on interfaces that insert fictitious queries and drop their replies without affecting the quality of the LBS servers' replies to real queries.

Chapter VII: Conclusion and Future Work

We summarize the main contributions of this dissertation and discuss the potential directions for future work.

1.3 Related Notions

1.3.1 Other Notions of Opacity

In [8] and [9], the authors have considered three notions of opacity: *initial opacity, final opacity*, and *total opacity* (or *always opacity* in [9]) in the modeling formalisms of Petri nets and labelled transition systems, respectively. The first two opacity notions are relevant to initial-state opacity and current-state opacity considered in this dissertation. Total opacity (or always opacity), on the other hand, requires all states in the execution to be hidden from the observer. It is relevant to current-state opacity when the set of secret states lie on a prefix-closed language.

Let us consider the modeling formalism of finite-state automata. The notion of *K-step opacity* has been introduced in [45]. It is a stronger version of current-state opacity that requires the secret to hold for K-delayed estimates of the state. The notions of *strong opacity* and *weak opacity* have been introduced in [33] and studied in [4, 5]. Both notions are defined in the language formulation. Strong opacity requires that *all* strings in a language be confused with some strings in another language, which is the same as our language-based opacity. Weak opacity, however, only requires that *some* strings in the language be confused with some strings in another language. On the other hand, the authors in [3] have considered the notion of *concurrent opacity.* This notion, similar to our notions of joint opacity, considers multiple intruders. But it differs from joint opacity in that each intruder in concurrent opacity has a distinct secret and observation map and that intruders do not collaborate.

1.3.2 Related Notions

Many security notions have also been defined in DES. In [26], the authors have considered the notion of *intransitive noninterference.* This notion is a confidentiality property in multilevel systems where information flow from the high level to the low level is only allowed through the downgrading level. The notion of *secrecy* has been considered in [1, 33]. It is a special notion of language-based opacity when the non-secret language is the complement of the secret language, with respect to the language of the given system.

Beyond the category of security properties, there are also notions that are relevant to opacity. We discuss the notions of observability, detectability, and diagnosability. Each of them is a *special negation of weak opacity* that requires strings of different features to be clearly distinguished. That is, observability, detectability, or diagnosability fails to hold if there exist two strings of different features that are not distinguishable. Note that these notions are *not* the negation of the opacity notions studied in this dissertation (i.e., they are not the negation of strong opacity) as opacity requires *all* secret behavior to be indistinguishable from a non-secret behavior. Specifically, *observability* requires two strings with different control actions, with respect to the specification, to be distinguishable [34]. *Detectability*, on the other hand, captures whether we can determine the current state of the system; it requires strings leading to different states to be distinguishable [54]. Lastly, the system is *diagnosable* if no string with fault events is indistinguishable from a string without fault events for an unbounded length [48]. We refer the reader to [33] for a detailed comparative discussion of these notions and opacity.

CHAPTER II

Opacity Problems in Automata Formulations

2.1 Automata Models and Relevant Operations

2.1.1 Automata Models

We consider opacity problems in DES systems modeled as (potentially nondeterministic) automata. An automaton $G = (X, E, f, X_0, X_m)$ has a finite set of states, a set of events E, a partial state transition function $f : X \times E \cup \{\varepsilon\} \to 2^X$, a set of initial states X_0, and a set of marked states X_m. In opacity problems, the initial state need not be known *a priori* by the intruder and thus we include a set of initial states X_0 in the definition of G. When there are no marked states, we write $G = (X, E, f, X_0)$. Set E^* contains all finite strings composed of elements of E. Function f is extended to domain $X \times E^*$ in a recursive manner: $f(x, t) = f(x, t'e) = f(f(x, t'), e)$ where $t' \in E^*$ and $e \in E$. Define $\mathcal{L}(G, X_i) := \{t \in E^* : (\exists x \in X_i)[f(x, t) \text{ is defined}\}$. The language generated by G is the system behavior that is defined by $\mathcal{L}(G, X_0)$. For simplicity, we write $\mathcal{L}(G, x)$ when $X_i = \{x\}$ and write only $\mathcal{L}(G)$ if X_0 is clearly defined. The marked language of G is defined by $\mathcal{L}_m(G, X_0) := \{t \in E^* : (\exists i \in X_0)[f(i, t) \cap X_m \neq \emptyset]\}$.

2.1.2 Projection Map P

The system is partially observable in general. Hence, the event set is partitioned into an observable set E_o and an unobservable set E_{uo}. Given an event $e \in E$, its observation is the output of the natural projection $P : E \to E_o$ such that $P(e) = e$ if $e \in E_o$ and $P(e) = \varepsilon$ if $e \in E_{uo} \cup \{\varepsilon\}$ where ε is the empty string. With this definition at hand, projection P is extended from $E \to E_o$ to $P : E^* \to E_o^*$ in a recursive manner: $P(te) = P(t)P(e)$ where $t \in E^*$ and $e \in E$.

2.1.3 Observer Automaton

Consider $G = (X, E, f, x_0)$ where $E = E_o \cup E_{uo}$. The observer automaton of G is the automaton $Obs(G) := (X_{obs}, E_o, f_{obs}, x_{0,obs})$ that is built as follows.

1. Define $x_{0,obs} := UR(x_0)$. Set $X_{obs} = \{x_{0,obs}\}$.
2. For each $B \in X_{obs}$ and $e \in E_o$, define $f_{obs}(B, e) := UR(\{x \in X : (\exists x_e \in B)[x \in f(x_e, e)]\})$. If $f(x_e, e)$ is defined for some $x_e \in B$, add state $f_{obs}(B, e)$ to X_{obs}.
3. Repeat Step 2 until the entire accessible part of $Obs(G)$ has been constructed

where $UR(x)$ is the *unobservable reach* of a state x that is defined to be the set of all states that can be reached from x via unobservable transitions.

2.1.4 Parallel Composition

Parallel composition captures the interaction of system components that synchronize on common events but act individually on private events. Consider $G_1 = (X_1, E_1, f_1, x_{01})$ and $G_2 = (X_2, E_2, f_2, x_{02})$. The parallel composition of G_1 and G_2 is the automaton $G_1||G_2 := Acc(X_1 \times X_2, E_1 \cup E_2, f, (x_{01}, x_{02}))$ where

$$f((x_1, x_2), e) :=$$
$$\begin{cases} \{(x_1', x_2') : x_1' \in f_i(x_i, e), i = 1, 2\}, \text{ if } e \in E_1 \cap E_2 \text{ and } f_i(x_i, e) \text{ is defined} \\ \{(x_1', x_2) : x_1' \in f_1(x_1, e)\}, \text{ if } e \in E_1 \setminus E_2 \text{ and } f_1(x_1, e) \text{ is defined} \\ \{(x_1, x_2') : x_2' \in f_1(x_2, e)\}, \text{ if } e \in E_2 \setminus E_1 \text{ and } f_2(x_2, e) \text{ is defined} \end{cases}$$

and Acc denotes the accessible part of the automaton, i.e., the set of states reachable from a initial state via some string $t \in (E_1 \cup E_2)^*$.

Finally, we refer to [10] for a tutorial of Discrete Event Systems.

2.2 Existing Notions of Opacity

We consider opacity properties in DES modeled as finite-state automata. The basic opacity problems assume the attack model with only one single intruder. The settings are: (1) G has a *secret*; (2) G is partially observable and/or nondeterministic; (3) the intruder is an observer of G that have full knowledge of the structure of G. Hence, the intruder, having the knowledge of G and the partial observation of G, can infer the real system behavior by constructing estimates. Opacity holds if no intruder's estimate reveals the occurrence of the secret. In other words, the system is opaque if *for any secret behavior, there exists another non-secret behavior that is observationally equivalent to the intruder.* Therefore, the intruder is never sure whether the secret has occurred or not.

Four notions of opacity studied in the literature will be focused in this dissertation: (i) Initial-State Opacity where the secret is a set of initial states; (ii) Current-State Opacity where the secret is a set of current states; (iii) Language-Based Opacity where the secret is a subset of the set of system runs. (iv) Initial-and-Final-State Opacity where the secret is a set of initial and finite states pairs. The first three have been studied in [42, 43, 11, 33] and will be defined in this section. IFO was introduced in our recent work [59] and will be defined in the next section.

2.2.1 Initial-State Opacity

The notion of initial-state opacity (or ISO) was first defined for Petri nets in [8], and then introduced to finite-state automata in [43]. Initial-state opacity property is a state property that relates to the membership of the system's initial state within a set of secret states. The system is initial-state opaque if the intruder is never sure whether the system's initial state is a secret state or not.

Definition II.1 (Initial-State Opacity)**.** Given system $G = (X, E, f, X_0)$, projection P, set of secret initial states $X_S \subseteq X_0$, and set of non-secret initial states $X_{NS} \subseteq X_0$, G is initial-state opaque if $\forall i \in X_S$ and $\forall t \in \mathcal{L}(G, i)$, $\exists j \in X_{NS}$, $\exists t' \in \mathcal{L}(G, j)$, such that $P(t) = P(t')$.

The system is initial-state opaque if for every string t that originates from secret state i, there exists another string t' from non-secret state j such that t and t' are observationally equivalent. Therefore, the intruder can never determine whether the system started from a secret state i or from a non-secret state j.

The following is the same example as in [43] that is used to demonstrate ISO. This example is used in this thesis in order to compare our algorithm with that in [43].

Example II.2. ([43]) Consider G in Figure 2.1 with $E_o = \{a, b\}, X_S = \{2\}$, $X_{NS} = X \setminus X_S$, and $X_0 = X$. G is initial-state opaque because for every string t starting from state 2, there is another string $(uo)t$ starting from state 0 that looks the same. However, ISO does not hold if $X_S = \{0\}$. Whenever the intruder sees string aa, it is sure that the system originated from state 0; no other initial state generates strings that look the same as aa.

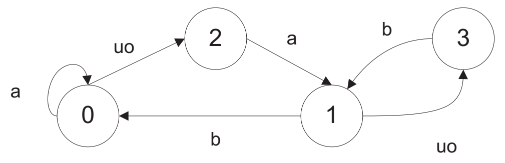


Figure 2.1: The system G discussed in Example II.2

2.2.2 Current-State Opacity

Current-state opacity was first introduced as "final opacity" in [8] in the context of Petri nets. The definition was then adopted in the framework of labelled transition systems in [9], and further developed in finite-state automata models [42, 11, 59]. A system is current-state opaque if the intruder can never infer, from its observations, whether the current state of the system is a secret state or not.

Definition II.3 (Current-State Opacity)**.** Given system $G = (X, E, f, X_0)$, projection P, set of secret states $X_S \subseteq X$, and set of non-secret states $X_{NS} \subseteq X$, the system is current-state opaque if $\forall i \in X_0$ and $\forall t \in \mathcal{L}(G, i)$ such that $f(i, t) \subseteq X_S$, $\exists j \in X_0$, $\exists t' \in L(G, j)$ such that: (i) $f(j, t') \cap (X \setminus X_S) \neq \emptyset$ and (ii) $P(t) = P(t')$.

Current-state opacity (or CSO) states that for every string t that leads to secret states only, there exists another string t' going to a non-secret state whose projection is the same. As a result, the intruder can never assert with certainty that the system's current state is in X_S. Note that the automaton model we use, where there can be multiple initial states, allows the two observationally equivalent strings to start from different initial states.

Example II.4. Consider G in Figure 2.2 with $X_S = \{3\}$ and $X_{NS} = X \setminus X_S$. Let $E_o = \{b\}$. G is current-state opaque because the intruder is always confused between ab and cb when observing b; that is, the intruder cannot tell if the system is in state 3 or 4. However, if $E_o = \{a, b\}$, then G is not opaque because the intruder is sure that the system is in state 3 when observing ab.

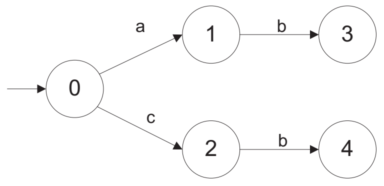


Figure 2.2: The system G discussed in Example II.4

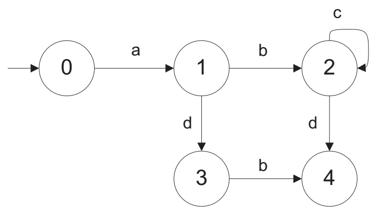


Figure 2.3: The system G discussed in Example II.6

2.2.3 Language-Based Opacity

Language-Based opacity (or simply LBO) was first defined using finite-state automata models in [18] and then further discussed in [19, 46, 11, 33, 59]. A system is language-based opaque if the intruder can never infer, from its observations, whether the current execution of the system is in the secret language or not.

Definition II.5 (Language-Based Opacity)**.** Given system $G = (X, E, f, X_0)$, projection P, secret language $L_S \subseteq \mathcal{L}(G, X_0)$, and non-secret language $L_{NS} \subseteq \mathcal{L}(G, X_0)$, G is language-based opaque if for every string $t \in L_S$, there exists another string $t' \in L_{NS}$ such that $P(t) = P(t')$, or equivalently if $L_S \subseteq P^{-1}[P(L_{NS})]$.

The system is language-based opaque if for every string $t \in L_S$, there exists at least one other string $t' \in L_{NS}$ with the same projection. Therefore, given the observation $s = P(t)$, intruders cannot conclude whether secret string t or non-secret string t' has occurred. Note that L_S and L_{NS} need not be prefix-closed in general. Also, they need not be regular; however, we assume that they are regular throughout this thesis.

Example II.6. Consider G in Figure 2.3 with $E_o = \{a, b, c\}$. It is language-based opaque when $L_S = \{abd\}$ and $L_{NS} = \{abc^*d, adb\}$ because whenever the intruder sees $P(L_S) = \{ab\}$, it is not sure whether string abd or string adb has occurred. However, this system is not language-based opaque if $L_S = \{abcd\}$ and $L_{NS} = \{adb\}$; no string in L_{NS} has the same projection as the secret string *abcd*.

2.3 New Notion of Opacity: Initial-and-Final-State Opacity

Initial-and-final-state opacity, first introduced in [59], is an opacity notion that models anonymous communications in security networks, where the identities of the sender and/or the receiver need to be hidden from the intruder [13, 40, 51]. Let us model a communication network as an automaton, and use the initial and the final states to represent the identities of the sender and the receiver. The network is anonymous if the memberships of the initial and the final states are hidden as a pair.

Definition II.7 (Initial-and-Final-State Opacity)**.** Given system $G = (X, E, f, X_0)$, projection P, set of secret state pairs $X_{sp} \subseteq X_0 \times X$, and set of non-secret state pairs $X_{nsp} \subseteq X_0 \times X$, G is initial-and-final-state opaque if $\forall (x_0, x_f) \in X_{sp}$ and $\forall t \in \mathcal{L}(G, x_0)$ such that $x_f \in f(x_0, t)$, there is a pair $(x_0', x_f') \in X_{nsp}$ and a string $t' \in \mathcal{L}(G, X_0)$ such that $x_f' \in f(x_0', t')$ and $P(t) = P(t')$.

Initial-and-final-state opacity (or simply IFO) requires that the occurrence of a string starting from x_0 and ending at x_f, where (x_0, x_f) is a secret pair, should not reveal to the intruder the fact that x_0 is the initial state *and* x_f is the final state. A system is said to be intial-and-final-state opaque if for any string t that starts from x_0 and ends at x_f, with $(x_0, x_f) \in X_{sp}$, there exists another string t' starting from x_0' and ending at x_f', where $(x_0', x_f') \in X_{nsp}$, that has the same projection. Therefore, the intruder can never determine whether the initial-and-final state pair is a secret pair or a non-secret pair.

Example II.8. Consider again G in Figure 2.1. G is initial-and-final state opaque when $X_{sp} = \{(3, 1)\}$ and $X_{nsp} = \{(1, 0), (1, 1), (1, 2), (1, 3)\}$. Now if we take $X_{sp} = \{(0, 0)\}$, then G is no longer initial-and-final-state opaque. In this case, $(0, 0)$ is the only state pair that corresponds to string aa; no other state pair gives strings that look the same as aa.

Remark II.9. ISO and CSO are both special cases of IFO. To obtain an ISO problem from an IFO problem, we set $X_{sp} = X_S \times X$ and $X_{nsp} = X_{NS} \times X$. Also, to obtain an CSO problem, we set $X_{sp} = X_0 \times X_S$ and $X_{nsp} = X_0 \times X_{NS}$.

Remark II.10. In the definitions of LBO, ISO, CSO, and IFO, no assumption is made regarding the sets of secret and non-secret languages, states, or state pairs, respectively. However, to facilitate our work, we can assume they are disjoint without loss of generality. Take ISO for example. Assume the intersecting set $X_I := X_S \cap X_{NS}$ is not empty. Then, every state in X_I is a secret state as well as a non-secret state. That is, a string from a secret state in X_I also comes from a non-secret state. Hence, no state in X_I results in the violation of opacity. ISO is unchanged if X_I is removed from the secret set X_S. Therefore, we can re-define the secret state set as $X'_S = X_S \setminus X_I$, which is disjoint with X_{NS}, without affecting ISO. Similar results hold for LBO, CSO and IFO.

CHAPTER III

Verification of Opacity in Centralized and Coordinated Architectures

3.1 Introduction

With the four notions of opacity defined, our first task is to develop algorithmic procedures that verify these opacity notions.

In this chapter, we start with investigating the relationships among the four notions of opacity. While these notions are formulated differently, we will show that each opacity notion can be reduced to other three opacity notions in polynomial complexity. This set of polynomial reduction algorithms establishes a verification algorithm for any given opacity notion by transforming it to another opacity notion.

Next, we discuss the most efficient verification algorithm for each opacity notion. Although we can always verify an given opacity by transforming it to another opacity notion, this method may not be the most efficient manner to proceed. We review existing methods for verifying opacity notions and propose a more efficient one for verifying ISO, which reduces the space complexity for verifying ISO from $O(2^{|X|^2})$ to $O(2^{|X|})$.

Finally, we extend the three state-based opacity notions (i.e., CSO, ISO, IFO) to a *coordinated architecture* where multiple intruders collude via a coordinator who integrates the state estimates from all intruders. We define three corresponding notions of *joint opacity* in the context of this coordinated architecture, and develop algorithmic procedures to verify the three joint opacity notions. To the best of our knowledge, this is the first study of opacity in architectures where intruders coordinate.

The remaining sections of this chapter are organized as follows. Section 3.2 presents the polynomial reduction procedures between the four notions of opacity. Section 3.3 considers the verification of the four opacity properties. Section 3.4 introduces and verifies opacity properties under the coordinated architecture. Section 3.5 discusses the adaptation of our results to the case of "fast" intruders, which react faster than the system can execute unobservable events. Finally, Section 3.6 concludes this chapter. Some of the results in this chapter also appear in [59].

3.2 Polynomial Reductions Between Four Notions of Opacity

While different notions of opacity have been defined in existing work, their relationships have never been completely characterized. To the best of our knowledge, the only works that consider such relationships are the alternative language-based definition for initial-state opacity in [46], the transformation from trace-based opacity (LBO in our terminology) to state-based opacity (CSO in our terminology) in [11], and the transformation in [33] from language-based opacity to strong-secrecy and weak-secrecy (defined therein). In this section, we provide a complete characterization of the relationships between the four notions of opacity defined in Chapter II. Algorithms that transform among the four notions are presented in the following subsections, according to the diagram in Figure 3.2. We provide complete proofs for the transformations between IFO and LBO in the corresponding subsections, and briefly discuss in Section 3.2.5 the proofs for the other transformations. For the sake of simplicity, we will use the acronym IFO to denote both "initial-and-final-state opacity" and "initial-and-final-state opaque"; similarly for LBO, CSO, and ISO. It will be clear from the context if the noun or the adjective is referred to.

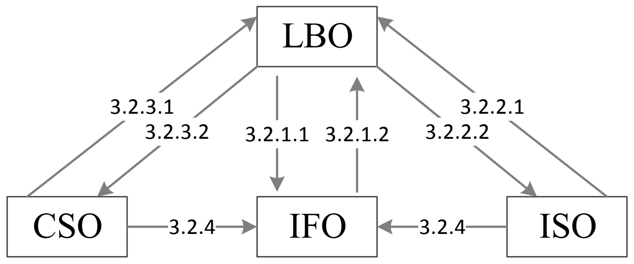


Figure 3.1: Transformations between notions of opacity (labeled by section numbers)

3.2.1 Transformation Between IFO and LBO

3.2.1.1 IFO to LBO

Given an IFO problem with $G = (X, E, f, X_0)$ and sets of secret and non-secret state pairs X_{sp} and X_{nsp}, we transform it to an LBO problem by the following steps:

1. Construct $G_i^s = Trim[(X, E, f, \{x_{0,i}^s\}, \{x_{f,i}^s\})]$ where $(x_{0,i}^s, x_{f,i}^s)$ is the i-th pair in X_{sp}, and $G_j^{ns} = Trim[(X, E, f, \{x_{0,j}^{ns}\}, \{x_{f,j}^{ns}\})]$ where $(x_{0,j}^{ns}, x_{f,j}^{ns})$ is the j-th pair in X_{nsp}.

2. Obtain G^s by treating the set of all G_i^s as a single automaton, with corresponding sets of initial and marked states. Proceed similarly to obtain G^{ns} from all G_j^{ns}.

3. Obtain G_{LBO} by treating G^s and G^{ns} as a single automaton. Define the secret language $L_S := \mathcal{L}_m(G^s) = \bigcup_i \mathcal{L}_m(G_i^s)$ and the non-secret language $L_{NS} := \mathcal{L}m(G^{ns}) = \bigcup_j \mathcal{L}_m(G_j^{ns})$.

We now show that (G, X_{sp}, X_{nsp}) is IFO if and only if (G_{LBO}, L_S, L_{NS}) is LBO. By construction, every G_i^s marks the language that corresponds to the i-th secret pair $(x_{0,i}^s, x_{f,i}^s) \in X_{sp}$. Since every secret pair has a corresponding G_i^s, language L_S captures the complete set of secret pairs X_{sp}. Similarly, L_{NS} captures the set of non-secret pairs X_{nsp}. To verify if (G, X_{sp}, X_{nsp}) is IFO, we check if every string with a secret pair $(x_0, x_f) \in X_{sp}$ has the same projection as a string associated with a non-secret pair $(x'_0, x'_f) \in X_{nsp}$, that is, if every string $t \in L_S$ has the same projection as a string $t' \in L_{NS}$ in G_{LBO}. This procedure is equivalent to verify if (G_{LBO}, L_S, L_{NS}) is LBO.

We discuss the computational complexity of this transformation. Given any input instance (G, X_{sp}, X_{nsp}), building G^s takes complexity $O(|X|^2|X_0|)$ because each G_i^s is simply the trim of G with the i-th state pair in X_{sp} as the initial and marked state, and there are at most $|X||X_0|$ number of such G_i^s. Similarly, building G^{ns} also takes complexity $O(|X|^2|X_0|)$. Therefore, this transformation can be obtained in polynomial time, in the cardinality of the state space of G.

3.2.1.2 LBO to IFO

Given an LBO problem with G, L_S, and L_{NS}, we construct the equivalent IFO problem by the following steps:

1. Build automaton $G^s = (X^s, E, f_s, X_0^s, X_f^s)$ with $\mathcal{L}_m(G^s, X_0^s) = L_S$ and automaton $G^{ns} = (X^{ns}, E, f_{ns}, X_0^{ns}, X_f^{ns})$ with $\mathcal{L}_m(G^{ns}, X_0^{ns}) = L_{NS}$.

2. Construct G_{IFO} by treating the two automata G^s and G^{ns} as a single one. Define the set of secret pairs $X_{sp} := X_0^s \times X_f^s$ and the set of non-secret pairs $X_{nsp} := X_0^{ns} \times X_f^{ns}$.

We show that (G, L_S, L_{NS}) is LBO if and only if $(G_{IFO}, X_{sp}, X_{nsp})$ is IFO. By construction, G^s, which has initial states X_0^s and marked states X_f^s, generates marked language L_S. Thus, a string is in L_S if and only if it has an initial-final state pair in $X_{sp} = X_0^s \times X_f^s$. Similarly, a string is in L_{NS} if and only if it has an initial-final state pair in $X_{nsp} = X_0^{ns} \times X_f^{ns}$. To verify if (G, L_S, L_{NS}) is LBO, we verify if every string in L_S is always confused with a string in L_{NS}; that is, if every state pair in X_{sp} is always confused with a state pair in X_{nsp}. This is the same as checking if

(G, X_{sp}, X_{nsp}) is IFO.

This transformation also requires polynomial complexity. In step 1, the complexity depends on how the languages L_S and L_{NS} are defined. Recall that L_S and L_{NS} are assumed to be regular. They could be specified using automata, in which case G^s and G^{ns} are directly obtained. More generally, L_S could be expressed as sublanguages of $\mathcal{L}(G)$ in terms of state and/or event ordering, in which case G^s can be obtained in polynomial time by splitting the state space of G as necessary using standard automata procedures. Step 2 takes only constant time. Therefore, the transformation can be done in polynomial time, in the cardinality of the state space of G.

Example III.1. Consider an LBO problem with the system in Figure 2.3. Let $L_S = \{abd\}$, $L_{NS} = abc^*d + adb$, and $E_o = \{a, b, c\}$. To transform the LBO problem into an IFO problem, we first build G^s (top in Figure 3.2) and G^{ns} (bottom in Figure 3.2). Then, we construct G_{IFO} by taking the two automata as a single one, as shown in Figure 3.2. Finally, we define the set of secret and non-secret pairs as $X_{sp} = \{(0, 3)\}$ and $X_{nsp} = \{(4, 7), (4, 9)\}$, respectively.

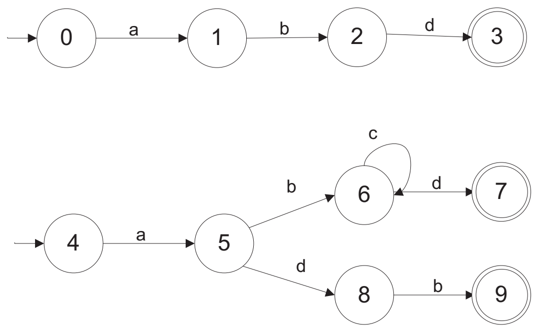


Figure 3.2: G_{IFO} used in Example III.1

3.2.2 Transformation Between ISO and LBO

3.2.2.1 ISO to LBO

Given an ISO problem with G, X_S, and X_{NS}, we construct the equivalent LBO problem by the following steps:

1. Build automata $G^s = Trim[G(X, E, f, X_S)]$ and $G^{ns} = Trim[G(X, E, f, X_{NS})]$.

2. Obtain G_{LBO} by treating G^s and G^{ns} as a single automaton. Define the secret and the non-secret languages as $L_S := \mathcal{L}(G_{LBO}, X_S)$ and $L_{NS} := \mathcal{L}(G_{LBO}, X_{NS})$.

Example III.2. Consider an ISO problem with the system in Figure 2.1. Let $X_S = \{0\}$ and $X_{NS} = X \setminus X_S = \{1, 2, 3\}$. We follow the above transformation and obtain secret and non-secret languages $L_S = \mathcal{L}(G, \{0\})$ and $L_{NS} = \mathcal{L}(G, \{1, 2, 3\})$. Because no string in L_{NS} looks the same as string $aa \in L_S$, we conclude that LBO is violated.

3.2.2.2 LBO to ISO

To transform from LBO to ISO, we must check if L_S and L_{NS} in the LBO problem are prefix-closed. This is because the languages of an ISO problem are prefix-closed by definition. If L_S and L_{NS} are not prefix-closed, then this transformation is not meaningful. Given an LBO problem with G, L_S, and L_{NS}, we now transform it to an ISO problem:

1. Check if L_S and L_{NS} are both prefix-closed. If yes, then proceed to Step 2. Otherwise, no transformation exists.

2. Build automaton $G^s = (X^s, E, f_s, X_0^s)$ and automaton $G^{ns} = (X^{ns}, E, f_{ns}, X_0^{ns})$ such that $\mathcal{L}(G^s, X_0^s) = L_S$ and $\mathcal{L}(G^{ns}, X_0^{ns}) = L_{NS}$.

3. Construct G_{ISO} by treating G^s and G^{ns} as a single automaton. Take the secret and non-secret initial states sets to be $X_S = X_0^s$ and $X_{NS} = X_0^{ns}$, respectively.

Example III.3. Consider again the system in Figure 2.3 with $E_o = \{a, b, c\}$. Let the secret and non-secret languages to be prefix-closed: $L_S = \overline{\{abd\}}$ and $L_{NS} = \overline{[abc^*d + adb]}$. This LBO problem can be transformed to an ISO problem with the system in Figure 3.2 (with no marked states) with secret state set $X_S = \{0\}$ and non-secret state set $X_{NS} = \{4\}$. In this example, ISO holds because for every string starting from state $\{0\}$, there is another string from $\{4\}$ with the same projection.

3.2.3 Transformation Between CSO and LBO

3.2.3.1 CSO to LBO

Given a CSO problem with G, X_S, X_{NS}, the equivalent LBO problem is built in two steps. First, we build automaton $G^s = Trim[G(X, E, f, X_0, X_S]$ and automaton $G^{ns} = Trim[G(X, E, f, X_0, X_{NS})]$. Then, we obtain G_{LBO} by treating G^s and G^{ns} as a single automaton, and define the secret and the non-secret languages as $L_S = \mathcal{L}_m(G_{LBO}, X_S)$ and $L_{NS} = \mathcal{L}_m(G_{LBO}, X_{NS})$.

Example III.4. Consider a CSO problem with the system in Figure 2.2. Let $X_S = \{3\}$, $X_{NS} = X \setminus X_S$, and $E_o = \{b\}$. We follow the above transformation and obtain secret and non-secret languages $L_S = \{ab\}$ and $L_{NS} = \{\varepsilon, a, c, cb\}$. In this case, LBO holds because secret string *ab* is observationally equivalent to non-secret string *cb*.

3.2.3.2 LBO to CSO

Given an LBO problem with G, L_S, L_{NS}, we transform it to a CSO problem in two steps. First, we build automaton $G^s = (X^s, E, f_s, X_0^s, X_f^s)$ and automaton $G^{ns} = (X^{ns}, E, f_{ns}, X_0^{ns}, X_f^{ns})$ such that $\mathcal{L}_m(G^s, X_0^s) = L_S$ and $\mathcal{L}_m(G^{ns}, X_0^{ns}) = L_{NS}$. Then, we construct G_{CSO} by treating G^s and G^{ns} as a single automaton, where the initial state set is $X_0 = X_0^s \cup X_0^{ns}$ and the secret and non-secret state sets are $X_S = X_f^s$ and $X_{NS} = X_f^{ns}$.

Example III.5. Consider an LBO problem with the system in Figure 2.3, $L_S = \{abd\}$, $L_{NS} = \{abc^*d, adb\}$, and $E_o = \{a, b, c\}$. We transform it to a CSO problem with the system in Figure 3.2, initial state set $X_0 = \{0, 4\}$, secret state set $X_S = \{3\}$, and non-secret state set $X_{NS} = \{7, 9\}$. CSO holds because string *adb*, which ends at state $\{9\}$, has the same projection as string *abd*, which is the only string that ends at secret state $\{3\}$.

3.2.4 Transformation Between ISO/CSO and IFO

As explained in Remark II.9, ISO and CSO are special cases of IFO, so the transformations from ISO to IFO and from CSO to IFO are already covered by that remark. On the other hand, the transformation from IFO to ISO can be obtained by first transforming IFO to LBO and then transforming LBO to ISO; similarly for the case IFO to CSO.

3.2.5 Discussion: Complexity

The transformations between ISO/CSO and LBO can be proven using similar methods for those between IFO and LBO. For the transformation from ISO to LBO, we construct G^s and G^{ns} whose initial states are X_S and X_{NS}, respectively. By suitably defining L_S and L_{NS} as sublanguages of G^s and G^{ns}, languages L_S and L_{NS} completely capture X_S and X_{NS}. Therefore, checking if a string starts from X_S or X_{NS} is equivalent to checking if the string is in L_S or L_{NS}. That is, verifying ISO in the original automaton G is equivalent to verifying LBO in G_{LBO}. A similar argument holds for the transformation from CSO to LBO. The only difference is that X_S and X_{NS} are the marked states of G^s and G^{ns}.

For the transformation from LBO to ISO, we construct G^s and G^{ns} that generate L_S and L_{NS} by suitably defining the initial states as X_S and X_{NS}. Therefore, checking if a string is in L_S or L_{NS} is equivalent to checking if the string starts from X_S or X_{NS}. That is, verifying LBO in the original automaton G is equivalent to verifying ISO in G_{ISO}. Similarly, by letting X_S and X_{NS} to be the marked states of G^s and G^{ns}, we can prove the transformation from LBO to CSO.

All the transformations are of polynomial-time computational complexity. We have seen that transformations between IFO and LBO require polynomial time. Transformations between ISO/CSO and LBO also require polynomial time because they are adapted from those between IFO and LBO. The transformations from IFO to ISO/CSO can also be done in polynomial time by transforming from IFO to LBO in polynomial time and then from LBO to ISO/CSO in polynomial time.

In conclusion, we have shown a polynomial-time transformation between any pair of the four opacity notions. The only exception is that there is no transformation from LBO to ISO when the secret or non-secret languages are not prefix-closed. Therefore, we can say that opacity notions OP_1 and OP_2 are "equivalent" in the sense that OP_1 holds in system G if and only if OP_2 holds in the the transformed system G'.

3.3 Verification of the Four Notions of Opacity

Based on the results of the preceding section, we can always verify one opacity notion by transforming it to another notion. However, this may not be the most efficient manner to proceed. In this section, we review existing verification methods for verifying ISO, CSO, and LBO, present a new algorithm for the verification of ISO that has reduced computational complexity as compared with existing ones, and present an algorithm for the verification of the new property of IFO.

Notice that there is no polynomial-time algorithm, in the cardinality of the state space of G, for verifying any of the above opacity notions. In [11], the authors have proven that verification of state-based opacity (CSO in our terminology) and that of trace-based opacity (LBO in our terminology) are PSPACE-complete problems; also, the authors in [41] has also proven that verifying ISO is PSPACE-complete. As a result, because of the polynomial-time transformations between IFO and other opacity notions in Section 3.2.5, verification of IFO must also be PSPACE-complete.

3.3.1 Verification of Initial-State Opacity

Verifying ISO can be done by capturing and examining all intruder's initial state estimates. Given that the intruder observes a string s, the intruder's initial state estimate corresponding to s is the set of initial states where a string t with observation $P(t) = s$ could have started. The formal definition for initial state estimates is given below:

Definition III.6. (Initial-state estimate) Given system $G = (X, E, f, X_0)$ and P, the initial state estimate after observing string s is defined as $\hat{X}_0(s) := \{i \in X_0 : (\exists t \in E^*)(P(t) = s)[f(i,t) \text{ is defined}]\}$

To capture all intruder's initial state estimates, the authors of [43] have constructed an *Initial State Estimator (ISE)* that captures initial state estimates based on trellis diagrams. Motivated by this ISE, we propose to construct a different ISE that captures initial state estimates by using the reversed automaton of the system. The reverse automaton will be formally defined later. To differentiate the two ISE, we call the former *Trellis-Based Initial State Estimator* and the latter *G_R-Based Initial State Estimator*. We start with a brief review of the trellis-based ISE.

3.3.1.1 Trellis-Based Initial State Estimator

The authors of [43] used state mappings to construct the trellis-based ISE. A state mapping $m \in 2^{X^2}$ is a subset of X^2 consisting of state pairs, which can be induced by an observed string. Specifically, given $s \in P[\mathcal{L}(G, X_0)]$, the induced state mapping $M(s)$ enumerates all possible pairs of starting and ending states corresponding to s. The composition operator for two state mappings $\circ : 2^{X^2} \times 2^{X^2} \to 2^{X^2}$ is defined as:

$$m_1, m_2 \in 2^{X^2}, m_1 \circ m_2 := \{(i_1, i_3) \mid \exists i_2 \in X, (i_1, i_2) \in m_1, (i_2, i_3) \in m_2\}$$

This composition operator takes the starting states of m_1 and the ending states of m_2 to form a new state mapping only for those sets of tuples that share the same intermediate element. Given G, the trellis-based ISE of G is a deterministic finite-state automaton where the state reached by string $s \in P[\mathcal{L}(G, X_0)]$ is state mapping $M(s)$. Note that the intruder is assumed to have no prior knowledge about the system's initial state. Hence, the ISE starts with a state mapping where the set of starting states is the entire state space X; more specifically, the initial state of the estimator is $\{(i,i) : i \in X)\} \cup \{(i,j) \in X^2 : (\exists t \in E_{uo}^*)[j \in f(i,t)]\}$. The transition function is defined such that state mapping m transitions to state mapping m' through event e_o if $m' = m \circ M(e_o)$. The ISE relies on state mappings to relay information about the initial and the current state estimates. It has been proven in [43] that the set of starting states of $M(s)$, i.e., the state mapping reached by s, is the intruder's initial state estimate $\hat{X}_0(s)$. One can verify ISO by examining all states in the trellis-based ISE and determining if any estimate contains only the secret states but not the non-secret states.

Remark III.7. In the above paragraph, we slightly modify the definition of the initial state of the trellis-based ISE from [43]. This modification allows us to verify IFO using the trellis-based ISE in Section 3.3.4.1 and to verify joint opacity notions in Section 3.4. Specifically, the modified initial state includes the unobservable reach to account for the unobservable transitions after the system begins and before the first observable transition. This is a practical concern as the intruder does not know when the system starts. The modification does not affect the initial state estimates since the unobservable reach affects only the ending states but not the starting states. Furthermore, it affects only the initial state but not the other states of the ISE. As a result, this modified trellis-based ISE gives the same initial state estimates as those in [43].

Example III.8. Consider the system in Figure 2.1. The a- and b-induced state mappings are $M(a) = \{(0,0),(0,1),(0,2),(0,3),(2,1),(2,3)\}$ and $M(b) = \{(1,0),(1,1),(1,2),(1,3),(3,1),(3,3)\}$. To construct the trellis-based ISE, we start with the initial state mapping m_0 as defined above. Then, we generate new state mappings by composing m_0 with $M(a)$, and $M(b)$. The complete construction is shown in Figure 3.3.

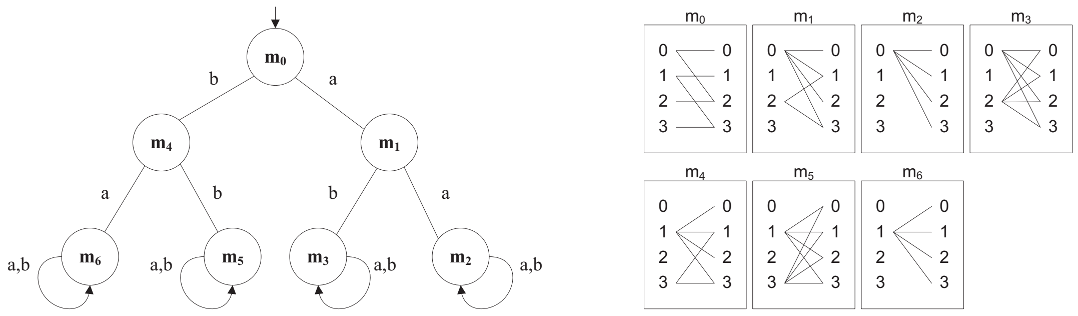


Figure 3.3: Trellis-based ISE of the system in Figure 2.1

3.3.1.2 G_R-Based Initial State Estimator

In the trellis-based ISE, the current state estimates need to be remembered in order to relay information while they are not involved in the verification of ISO. Because the trellis-based ISE keeps track of both the initial and the current state estimates, its construction has complexity $O(2^{|X|^2})$. However, there are at most $2^{|X|}$ possible initial state estimates. In this regard, we propose to construct a different initial state estimator based on the reversed automaton. This ISE, called the G_R-*based ISE*, contains only $O(2^{|X|})$ states. In the following, we will first define the reversed automaton, and then present the algorithm for verifying ISO using the G_R-based ISE.

The reversed automaton of G, denoted by G_R, is constructed by reversing all transitions in G. Given a string t, the reverse operator $Rev : E^* \to E^*$ outputs a new string t_R with events in the reversed order. Formally, the reversed automaton of G is defined as follows.

Definition III.9 (Reversed automaton G_R)**.** Given $G = (X, E, f, X_0, X_m)$, the reversed automaton G_R is the nondeterministic automaton that is obtained by reversing all the transitions in G. Specifically, $G_R := (X, E, f_R, X_{R,0}, X_{R,m})$ where $f_R(x', e) = \{x \in X : x' \in f(x, e)\}$, and $X_{R,0}$ and $X_{R,m}$ are to be specified according to the context.

Similarly, f_R is extended to domain $X \times E^*$ in a recursive manner: $f_R(x', se) = \{x \in X : (\exists x'' \in X)[x \in f_R(x'', e), x'' \in f_R(x', s)]\}$; or equivalently, $f_R(x', se) = \{x \in X : x' \in f(x, es_R)\}$.

With the reversed automaton defined, we now verify ISO. The verification will be discussed in two cases: (i) $X_0 = X$ when the intruder has no prior knowledge of the initial states, and (ii)$X_0 \subset X$ when the intruder has some prior knowledge of the initial states.

Case 1: Verification of ISO when $X_0 = X$

To construct the G_R-based ISE, we first build the reversed automaton G_R of the system G. Then, we build the observer $Obs(G_R, X)$, with the set of initial states being the entire state space X. We will prove below that the state of $Obs(G_R, X)$ reached by string s_R is the initial state estimate after s, where s_R is the reversed string of s. Hence, the intruder, with no prior knowledge of the system's initial states, has an initial state estimate being X when it has not observed anything. This is why we take the initial state as X.

We first present three lemmas that characterize useful properties of reversed strings and automata. A string with subscript R is the corresponding reversed string; that is, t_R is the reversed string of t. Operation $Rev(\cdot)$ is used to take the reverse of a string or the reverse of all the strings in a set of strings.

Lemma III.10. *$P(t_R) = P(t'_R)$ iff $P(t) = P(t')$.*

Proof. This is proved in a straightforward manner by induction on the length of $P(t)$. $\square$

Lemma III.11. *$x_0 \in f_R(x, t_R)$ iff $x \in f(x_0, t)$.*

Proof. This is proved using the extended definition of f_R; that is, $f_R(x, t_R) = \{x_0 \in X : x \in f(x_0, t)\}$. $\square$

Lemma III.12. *$t_R \in \mathcal{L}(G_R, X)$ iff $t \in \mathcal{L}(G, X)$. Thus, $\mathcal{L}[Obs(G_R, X)] = Rev\big(P[\mathcal{L}(G, X)]\big)$.*

Proof.

$$\begin{aligned} t \in \mathcal{L}(G, X) &\Leftrightarrow \exists x, x' \in X, \text{ such that } x' \in f(x, t) \\ &\Leftrightarrow \exists x, x' \in X, \text{ such that } x \in f_R(x', t_R) \text{ by Lemma III.11} \\ &\Leftrightarrow t_R \in \mathcal{L}(G_R, X) \end{aligned}$$

Furthermore, because $t_R = Rev(t)$, we have $\mathcal{L}(G_R, X) = Rev[\mathcal{L}(G, X)]$. By applying projection operation at both sides, we obtain $P[\mathcal{L}(G_R, X)] = P\big(Rev[\mathcal{L}(G, X)]\big)$; that is, $\mathcal{L}[Obs(G_R, X)] = Rev\big(P[\mathcal{L}(G, X)]\big)$. $\square$

We now present in Theorem III.13 how initial state estimates can be obtained from $Obs(G_R, X)$.

Theorem III.13. *Given $G = (X, E, f, X)$, P, $X_S \subseteq X$, and $X_{NS} \subseteq X$, the initial state estimate after observing $s \in P[\mathcal{L}(G, X)]$ is $\hat{X}_0(s) = f_{obs,R}(X, s_R)$, where $f_{obs,R}$ is the transition function of $Obs(G_R, X)$.*

Proof. For any string $s \in P[\mathcal{L}(G, X)]$, let us pick a state $x_0 \in \hat{X}_0(s) := \{i \in X : (\exists t \in E^*)(P(t) = s)[f(i, t) \text{ is defined}]\}$. Then we have that $\exists t \in \mathcal{L}(G, X)$ where $P(t) = s$ such that $f(x_0, t)$ is defined. Using Lemma III.12, we have that $\exists t \in \mathcal{L}(G, X) \Leftrightarrow \exists t_R \in \mathcal{L}(G_R, X)$. Using Lemma III.11, we have that $x \in f(x_0, t) \Leftrightarrow x_0 \in f_R(x, t_R)$. Using Lemma III.10, we have that $P(t) = s \Leftrightarrow P(t_R) = s_R$. Therefore,

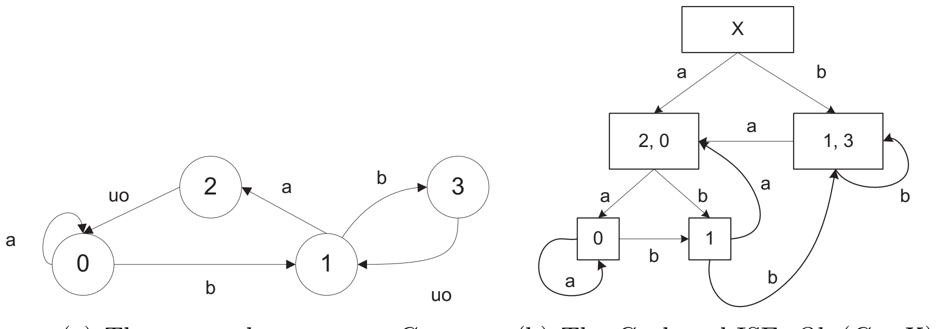
$x_0 \in \hat{X}_0(s)$ if and only if $\exists t_R \in \mathcal{L}(G_R, X)$, $\exists x \in X$ where $P(t_R) = s_R$ and $x_0 \in f_R(x, t_R)$. Equivalently, $x_0 \in \{i \in X : (\exists x \in X)(\exists t_R \in \mathcal{L}(G_R, X))[P(t_R) = s_R] \wedge [i \in f_R(x, t_R)]\} =: f_{obs,R}(X, s_R)$. □

Theorem III.13 shows that the intruder's initial state estimate after observing string s is captured by the state of $Obs(G_R, X)$ reached by s_R. Hence, $Obs(G_R, X)$ can be used as an ISE. In fact, we call this ISE the *G_R-based ISE* as it is built from the reversed automaton. One can verify ISO by examining all states in the G_R-based ISE. The system is not ISO if there exists an observation sequence that leads to an initial state estimate containing secret states but not non-secret states, as formalized by the following result. We use $X_{obs,R}$ to denote the state space of $Obs(G_R, X)$.

Theorem III.14. *$G = (X, E, f, X)$ is ISO if and only if $\forall y \in X_{obs,R}$, $y \cap X_S \neq \emptyset \Rightarrow y \cap X_{NS} \neq \emptyset$.*

Proof. By definition, G is ISO if and only if the system's initial state estimate never contains secret states but not non-secret states. Whenever there is a secret state in the estimate, there must also be a non-secret state to confuse the intruder, for all observable strings $s \in P[\mathcal{L}(G, X)]$. To prove the result it is sufficient to prove that the collection of all possible initial state estimates of the intruder equals to the collection of all reachable states of $Obs(G_R, X)$. We first observe that each initial state estimate is captured by a state in $Obs(G_R, X)$. This is given in the result of Theorem III.13 where $\hat{X}_0(s) = f_{obs,R}(X, s_R)$ for all $s \in P[\mathcal{L}(G, X)]$. For the reverse direction, we observe that each reachable state of $Obs(G_R, X)$ corresponds to a valid initial state estimate. This is because $Obs(G_R, X)$ is deterministic and $\mathcal{L}[Obs(G_R, X)] = Rev\big(P[\mathcal{L}(G, X)]\big)$ by Lemma III.12. Therefore, ISO can be verified by examining all reachable states of $Obs(G_R, X)$. □

Example III.15. Let us revisit the system in Figure 2.1 and use the G_R-based ISE to verify ISO. To build the G_R-based ISE, we first build G_R by reversing all the transitions in G, as shown in Figure 3.4(a). Then, we build $Obs(G_R, X)$ in Figure 3.4(b). Recall that the initial state is X because the intruder has no prior knowledge of the system's initial state. The state reached by string t_R is the intruder's initial state estimate after observing t. In this example, after observing event b, the intruder constructs initial state estimate $\{1, 3\}$, which is the state of the ISE reached by $Rev(b) = b$. If the intruder further observes a, its initial state estimate is updated to $\{1\}$, which is the state of the ISE reached by $Rev(ba) = ab$. To verify the ISO property, we examine all the states of the G_R-based ISE in Figure 3.4(b). If the secret state is $\{1\}$, the system is not ISO because the state of the ISE reached by ab contains only state 1.



(a) The reversed automaton G_R (b) The G_R-based ISE, $Obs(G_R, X)$

Figure 3.4: Construction of the G_R-based ISE of the system in Figure 2.1

Case 2: Verification of ISO when $X_0 \subset X$

In Case 1, we verified ISO when $X_0 = X$. Now, we consider the case when $X_0 \subset X$, which was mentioned but not studied in [43].

Recall that when $X = X_0$, the language of the G_R-based ISE is $\mathcal{L}[Obs(G_R, X)] = Rev(P[\mathcal{L}(G, X)]) = Rev(P[\mathcal{L}(G, X_0)])$; thus, the set of initial state estimates is the set of reachable states of $Obs(G_R, X)$. However, when $X_0 \subset X$, we could have $Rev(P[\mathcal{L}(G, X_0)]) \subset Rev(P[\mathcal{L}(G, X)]) = \mathcal{L}[Obs(G_R, X)]$. In this case, there exists a string t_R in $\mathcal{L}[Obs(G_R, X)]$ but not in $Rev(P[\mathcal{L}(G, X_0)])$. Thus, t_R does not correspond to a valid initial state estimate; namely, state $y = f_{obs,R}(X, t_R) \in$

$X_{obs,R}$ does not give a valid initial state estimate. In this case, we need to identify and examine only valid initial state estimates instead of examining all reachable states of $Obs(G_R, X)$. Marking of states is used for this purpose. First, we construct a modified automaton G' by marking all states in X_0 to recognize all valid initial states. Then, we reverse the transitions in G' and build the reversed automaton $G'_R = (X, E, f_R, X, X_0)$. Finally, the G'_R-based ISE is $Obs(G'_R, X) = (X_{obs,R}, E_o, f_{obs,R}, X, X_{obs,R,m})$. Note that the marked language of the G'_R-based ISE is the reversed projection language, i.e., $\mathcal{L}_m[Obs(G'_R, X)] = Rev\big(P[\mathcal{L}(G, X_0)]\big)$. Every marked state corresponds to a reversed string that starts from a valid initial state. To obtain the valid initial state estimate from a given marked state of $Obs(G'_R, X)$, we take the intersection of the marked state and X_0. (Note that an observer state is marked if it contains at least one marked state.) In Theorem III.16, we show how we obtain the desired initial state estimate from the G'_R-based ISE.

Theorem III.16. *Consider $G = (X, E, f, X_0)$ with $X_0 \subset X$, projection P, set of secret states $X_S \subset X$, and set of non-secret states $X_{NS} \subset X$. For every $s \in P[\mathcal{L}(G, X_0)]$, there exists $y_m = f_{obs,R}(X, s_R) \in X_{obs,R,m}$ such that the initial state estimate $\hat{X}_0(s)$ is $y_m \cap X_0$.*

Before proving Theorem III.16, we first characterize the properties of $Obs(G'_R, X)$ in the following lemmas.

Lemma III.17. *$f(x_0, t)$ is defined if and only if $x_0 \in f_{obs,R}(X, s_R) \cap X_0$, where $P(t_R) = s_R$.*

Proof. Consider that $f(x_0, t)$ is defined. We know that there exists $x \in X$, such that $x \in f(x_0, t)$. That is, $\exists x \in X, x \in f(x_0, t)$ and $x_0 \in X_0$ because $x_0 \in X_0$ is always true. According to Lemma III.11, the above condition is equivalent to $\exists x \in X, x_0 \in f_R(x, t_R) \wedge x_0 \in X_0$, which says $\exists x \in X, x_0 \in f_R(x, t_R) \cap X_0$. Finally, $\exists x \in X, x_0 \in f_R(x, t_R) \cap X_0$ means that, when we consider the observer, $x_0 \in f_{obs,R}(X, s_R) \cap X_0$

where $P(t_R) = s_R$. As a result, starting from the left-hand side, we have proven that $f(x_0, t)$ is defined if and only if $x_0 \in f_{obs,R}(X, s_R) \cap X_0$, where $P(t_R) = s_R$. $\square$

Lemma III.18. *$t \in \mathcal{L}(G, X_0)$ if and only if $t_R \in \mathcal{L}_m(G'_R, X)$.*

Proof.

$$\begin{aligned} t \in \mathcal{L}(G, X_0) &\Leftrightarrow \exists x_0 \in X_0, \exists x \in X, \text{ such that } x \in f(x_0, t) \\ &\Leftrightarrow \exists x_0 \in X_0, \exists x \in X, \text{ such that } x_0 \in f_R(x, t_R) \text{ by Lemma III.11.} \\ &\Leftrightarrow t_R \in \mathcal{L}_m(G'_R, X) \text{ because } X_0 \text{ is the set of marked states of } G'_R \end{aligned}$$

$\square$

We can now prove Theorem III.16.

Proof. Consider a string $s \in P[\mathcal{L}(G, X_0)]$. Let us pick a state $x_0 \in \hat{X}_0(s) := \{i \in X_0 : (\exists t \in E^*)(P(t) = s)[f(i, t) \text{ is defined}]\}$. There is $t \in \mathcal{L}(G, X_0)$ where $P(t) = s$ such that $f(x_0, t)$ is defined. Using Lemma III.18, we have that $t \in \mathcal{L}(G, X_0) \Leftrightarrow t_R \in \mathcal{L}_m(G'_R, X)$. Using Lemma III.10, we have that $P(t) = s \Leftrightarrow P(t_R) = s_R$. Using Lemma III.17, we know that $f(x_0, t)$ is defined $\Leftrightarrow x_0 \in f_{obs,R}(X, s_R) \cap X_0$. Consequently, $x_0 \in \hat{X}_0(s)$ if and only if $\exists x \in X$, $\exists t_R \in \mathcal{L}_m(G'_R, X)$ where $P(t_R) = s_R \in \mathcal{L}_m[Obs(G'_R, X)]$, such that $x_0 \in f_{obs,R}(X, s_R) \cap X_0$. That is, $x_0 \in f_{obs,R}(X, s_R) \cap X_0$ where $s_R \in \mathcal{L}_m[Obs(G'_R, X)]$, which proves Theorem III.16. $\square$

Theorem III.16 gives the initial state estimate of an intruder that has prior knowledge of $X_0 \subset X$. The intruder's estimate after observing string s is the intersection of X_0 and the marked state of $Obs(G_R, X)$ reached by s_R. To verify ISO when $X_0 \subset X$, we need to examine all marked states of $Obs(G_R, X)$.

Theorem III.19. *$G = (X, E, f, X_0)$ is ISO if and only if $\forall y_m \in X_{obs,R,m}$, $(y_m \cap X_0) \cap X_S \neq \emptyset \Rightarrow (y_m \cap X_0) \cap X_{NS} \neq \emptyset$.*

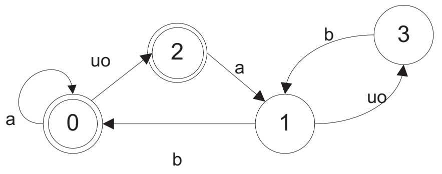
Proof. To prove Theorem III.19, it is sufficient to prove that the collection of all initial state estimates is equal to the collection of $Z := \{z = y_m \cap X_0 : y_m \in X_{obs,R,m}\}$. By Theorem III.16, we know that every initial state estimate is captured by the intersection of X_0 and a marked state of the G'_R-based ISE. That is, every initial state estimate is inside set Z. As for the other direction, because $\mathcal{L}_m[Obs(G'_R, X)] = Rev(P[\mathcal{L}(G, X_0)])$, we always obtain a valid initial state estimate by intersecting a marked state of the G'_R-based ISE with X_0. That is, every element in Z is an initial state estimate. This completes the proof. $\square$

Corollary III.20. *The computational complexity of verifying ISO using the G_R-based ISE in Theorem III.14 and Theorem III.19 is $O(2^{|X|})$.*

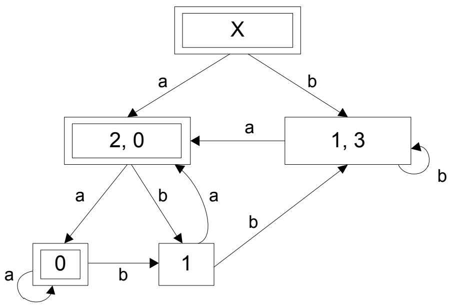
Corollary III.20 states the advantage of using the G_R-based ISE to verify ISO. The trellis-based ISE in [43] has complexity $O(2^{|X|^2})$ because of the use of state mappings as building blocks for the ISE.

Example III.21. Consider again the system in Figure 2.1 with $X_S = \{0\}$, and $X_{NS} = \{2\}$. Suppose $X_0 = \{0, 2\}$ and that the intruder has prior knowledge of X_0. To verify if the system is ISO, we construct the modified automaton G' by marking all initial states, as shown in Figure 3.5(a), and then construct the G'_R-based ISE, as shown in Figure 3.5(b). Notice that only states 0 and 2 are valid initial states of the system. Hence, not all states but only marked states reached in G'_R-based ISE correspond to valid initial state estimates. For example, state $\{1, 3\}$ is not a valid initial state estimate; no string starting from 0 or 2 has its reversed observable string that reaches state $\{1, 3\}$ in the G'_R-based ISE. To verify ISO, we consider all marked states of G'_R-ISE and intersect them with X_0. The system is not ISO because marked state $\{0\}$ is a valid initial state estimate that contains only secret initial state.

We conclude this section with two remarks that apply to both Case 1 and Case 2.



(a) The modified automaton G' where initial states 0 and 2 are marked



(b) The G'_R-based ISE: $Obs(G'_R, X)$

Figure 3.5: Verifying ISO when $X_0 \subset X$

Remark III.22. Consider Case 1; a similar argument holds for Case 2. A state of the G_R-based ISE reached by string s_R is the initial state estimate after observing s. In fact, the state represents only the state estimate after observing s; it does not possess any physical meaning if viewed as an intermediate state or as the starting state of another string. For example, the G_R-based ISE starts from X, the state reached by ε, because the intruder's initial state estimate after observing nothing is $\hat{X}_0(\varepsilon) = X$. If event a is observed, then the intruder's estimate moves to $\hat{X}_0(a) = f_{obs,R}(X, a)$, which is the state reached by $a = Rev(a)$. Although $\hat{X}_0(a)$ is reached from $\hat{X}_0(\varepsilon)$, $\hat{X}_0(\varepsilon)$ is not the final state estimate after observing a. Constructed in a reversed manner, states of G_R-based ISE along the same path share the same suffix but not prefix in general; thus, those state estimates are not the intermediate or final state estimate of one another in general. While the G_R-based ISE does not show the evolution of the intruder's initial state estimates, it is sufficient for verifying ISO because it provides the set of all initial state estimates.

Remark III.23. The construction of the G_R-based ISE does not assume a specific set of secret states. The set X_S is not considered until we verify ISO and examine all (marked) states of the G_R-based ISE. As a result, if X_S changes, one does not need to reconstruct the ISE, but only need to test the inclusion relationship with the new X_S to verify ISO.

3.3.2 Verification of Current-State Opacity

The most intuitive way to verify CSO is to build the observer automaton. Given G, the observer $Obs(G, X_0)$ models how the intruder gains knowledge of the system through observations. More specifically, the state of $Obs(G, X_0)$ reached by $s \in P[\mathcal{L}(G, X_0)]$ is the intruder's state estimate after observing s. Therefore, we can use $Obs(G, X_0)$ to capture all possible state estimates of the intruder. To verify CSO, we examine all reachable states in $Obs(G, X_0)$. The system is CSO if no state in $Obs(G, X_0)$ contains secret states but not non-secret states. Also, when constructing the observer to verify CSO, one does not assume a specific set of secret states. Thus, no reconstruction of the observer is required if the set of secret states changes.

3.3.3 Verification of Language-Based Opacity

Given $L_S,\ L_{NS} \subseteq \mathcal{L}(G, X_0)$, the system is LBO if $L_S \subseteq P^{-1}[P(L_{NS})]$. To check the aforestated language inclusion, the author of [33] proposed Algorithm 1 therein that utilizes marked languages of automata. We use the notation of [33] to briefly review that algorithm for the sake of comparison with a transformation-based approach. In order to fit our model, Algorithm 1 of [33] needs to be slightly modified by using the natural projection instead of the more general state-based projection. In brief, the algorithm first constructs two automata, G_1 and G_2, that mark L_S and L_{NS}, respectively. Then, it constructs their observers, G_5 and G_8, respectively. Next, it builds $G_9 := G_5 \times G_8$, which marks the joint projected marked language. Finally, it compares $\mathcal{L}_m(G_9)$ and $\mathcal{L}_m(G_5)$. The system is LBO if $\mathcal{L}_m(G_9) = \mathcal{L}_m(G_5)$. That is, the system is LBO if $P(L_S) \cap P(L_{NS}) = P(L_S)$, or equivalently, $P(L_S) \subseteq P(L_{NS})$. As an alternative, we can use state-based verification by transforming the LBO problem to a CSO one. The state-based verification is based on the transformation from LBO to CSO presented in Section 3.2.3.2. The equivalent CSO problem is then verified as described above in Section 3.3.2. While the above two algorithms are constructed differently, they have the same computational complexity. In Algorithm 1, if we assume that G_1 and G_2 have state spaces in the order of X, then building G_9 has worst-case complexity of $O(2^{2|X|})$. As for state-based verification, transforming from LBO to CSO doubles the state space to $2X$ and building the observer also has worst-case complexity of $O(2^{2|X|})$.

3.3.4 Verification of Initial-and-Final-State Opacity

The notion of IFO considers the memberships of both the initial and the final states of the system. To verify IFO, one needs to construct the initial-and-final state pair estimates that correspond to the intruder's knowledge. We propose use the trellis-based ISE to verify IFO when the set of secret pairs X_{sp} and the set of non-secret pairs X_{nsp} are any arbitrary subsets of X^2. When X_{sp} and X_{nsp} are given in the form of Cartesian products, we propose use the G_R-based ISE or the observer to verify IFO.

3.3.4.1 Verification of IFO: General Case

Let us first consider the case where X_{sp} and X_{nsp} are any arbitrary subsets of X^2. In this case, enumeration of all possible starting and ending state pairs is needed to verify IFO. The trellis-based ISE in [43] gives such an enumeration by using state mappings, as described earlier. We propose use it in a straightforward manner to verify IFO. For all reachable states of the trellis-based ISE, if secret pairs always come along with non-secret pairs, then the system is IFO; otherwise, it is not IFO.

Example III.24. Let us go back to the system in Figure 2.1 and take the sets of secret and non-secret state pairs to be $X_{sp} = \{(0,0)\}$ and $X_{nsp} = \{(0,1),(0,2),(0,3)\}$. To model the intruder, we use the trellis-based ISE in Figure 3.3, which generates a set of state pairs for each observed string. In this example, the intruder will construct initial-and-final state estimate $m_1 = \{(0,0),(0,1),(0,2),(0,3),(2,1),(2,3)\}$ if it observes a.

Then, if the intruder observes another a (i.e, it has observed aa), it would update its estimate to $m_2 = \{(0,0),(0,1),(0,2),(0,3)\}$. To verify IFO, we examine all reachable states of this ISE. Because whenever there is the secret state pair $(0,0)$, we always have a non-secret state pair, the system is IFO.

3.3.4.2 Verification of IFO: X_{sp} and X_{nsp} in Cartesian Product Form

Verification of IFO can be simplified when both X_{sp} and X_{nsp} are expressed as Cartesian products. In this special case, it is not necessary to remember the exact initial and final state pair. It is sufficient to remember whether the initial and the final states are secret or not. Consequently, the G_R-based ISE or the standard observer can be used to verify IFO. We write $X_{sp} = X_0^s \times X_f^s$ and $X_{nsp} = X_0^{ns} \times X_f^{ns}$, and use $X_0^s, X_0^{ns}, X_f^s, X_f^{ns}$ as alternative parameters for the problem.

Verifying IFO Using the G_R-Based ISE

Given system G, the procedure to follow is:

1. Label states in X_f^s with S and states in X_f^{ns} with N by right-concatenating the label with the state name. (Right concatenation indicates that the labels are used for final states)

2. Build the G_R-based ISE. When constructing the observer, pass the label such that the successor carries the label of the predecessor.

3. The system is IFO if for every state containing i_0S where $i_0 \in X_0^s$, it also contains j_0N where $j_0 \in X_0^{ns}$.

Indeed, IFO problem in this special case can be thought of as an ISO problem with marked states, which considers the ISO property with respect to a marked language. In this case, the languages of the ISO problem are not prefix-closed in general. We did not discuss this case previously in order to keep the formulation of ISO problems simple. If a system is marked-state initial-state opaque, then for every string starting from X_S and ending at a marked state in X_m, there is another string starting from X_{NS} and ending X_m with the same projection. That is, every state pair in $X_S \times X_m$ is confused with some state pair in $X_{NS} \times X_m$, which is an IFO problem in Cartesian product form.

Verifying IFO Using the Observer Automaton

Similarly, we can verify the IFO problem in Cartesian product form using observers. First, label secret and non-secret *initial states* by left-concatenating S or N. (Left concatenation indicates that the labels are used for the initial-states). Then, build the observer and pass the label as before. The system is IFO if for every state containing Si_f where $i_f \in X_f^s$, it also contains Nj_f where $j_f \in X_f^{ns}$.

Remark III.25. When we use the trellis-based ISE to verify IFO, X_{sp} and X_{nsp} are not considered in the construction of the trellis. One can verify an IFO problem that has different X_{sp} and/or X_{nsp} without reconstructing the trellis-based ISE. However, our methods for verifying IFO when X_{sp} and X_{nsp} are in Cartesian product form are less flexible. Specifically, to use the G_R-based ISE, the final state sets X_f^s and X_f^{ns} need to be fixed; to use the observer automaton, the initial state sets X_0^s and X_0^{ns} need to be fixed. While the latter two methods have only one degree of freedom in varying state sets, their complexity is lower compared to that of the trellis-based ISE. Using the G_R-based ISE or the observer has complexity $O(2^{2|X|})$, while using the trellis-based ISE has complexity $O(2^{|X|^2})$.

3.3.5 Discussion: No Polynomial Algorithm for Verifying Opacity

In this section, we compare the verification of diagnosability and that of opacity. The notion of diagnosability is often informally considered as the negation of opacity, which requires faulty strings to be distinguishable from non-faulty strings in a bounded length. However, diagnosability can be verified in polynomial time [64] while verification of opacity is PSPACE-complete [11, 41]. One interesting question is why opacity cannot be verified in polynomial time.

A short answer to this question is that the quantifiers in the two notions are different. With no loss of generality, let us consider language-based opacity for the purpose of discussion. Opacity requires that *for all* secret strings, *there exists* a non-secret string that is observationally equivalent. On the other hand, non-diagnosability (i.e., the negation of diagnosability) is that *there exists* a faulty string, *there exists* a non-faulty string, both of arbitrarily long length and are observationally equivalent. Consequently, to verify non-diagnosability, on can synchronize all faulty strings and all non-faulty strings. If the synchronization contains strings of arbitrary long length, then we know there is a pair of arbitrarily-long faulty and non-faulty strings that are indistinguishable; hence, the system is non-diagnosable. However, to verify opacity, we first synchronize all secret and non-secret strings. After the synchronization, the "for all" requires us to also check if the resulting synchronization contains *all* the secret strings. This check is to check language equivalence of two nondeterministic automata (because of partial observation), which is why there is no polynomial algorithm for verifying opacity.

Note that, in [33], the author has proven that non-diagnosability is equivalent to weak opacity, which is a weaker notion of opacity that requires only *some* secret strings to have a corresponding observationally equivalent non-secret string. Verifying this notion of weak opacity can be done in polynomial time [65].

3.4 Joint Opacity Notions in the Coordinated Architecture

3.4.1 The Coordinated Architecture

In this section, we extend the study of the three state-based opacity properties (i.e., ISO, CSO, and IFO) to a coordinated architecture where intruders work as a team to infer the secret. In the spirit of [16], we study a simplified coordinated architecture where two local intruders communicate with one coordinator, as shown in Figure 3.6. Each local intruder knows the system model. They observe the system through their individual projection map, generate local state estimates, and then report their estimates to the coordinator. The coordinator has no knowledge of the system. It forms the so-called *coordinated estimate* by taking the intersection of the local estimates it receives. The communication from the local intruders to the coordinator is assumed to have no delay. The collaboration is restricted by the following rules: (i) local intruders do not communicate with each other about their individual estimates; (ii) local intruders have no knowledge of the projection map of one another; and (iii) the only collaboration between the two local intruders is through the coordinator, whose only available memory is to store the most recent coordinated estimate. The system is said to be *jointly opaque* if no coordinated estimate ever reveals the secret information. Because of the restricted collaboration, the coordinated estimate is no finer than the estimate of a *single* "system intruder" that would observe all events that are observable to some intruder. Such coordinated structures capture situations where a system intruder does not exist and where the coordination among local intruders is restricted.

In the next three sections, we define and verify the joint notions of ISO, CSO, and IFO that correspond to this specific coordinated architecture. As before, the system is modeled as a finite-state automaton $G = (X, E, f, X_0)$. But this time there are two sets of observable events, $E_{o,1}$ and $E_{o,2}$, one for each intruder, with respective sets of unobservable events $E_{uo,1}$ and $E_{uo,2}$ and associated natural projections P_1 and P_2. We denote by E_o the union $E_{o,1} \cup E_{o,2}$, and let E_{uo} be the set of unobservable events corresponding to E_o and P be the natural projection.

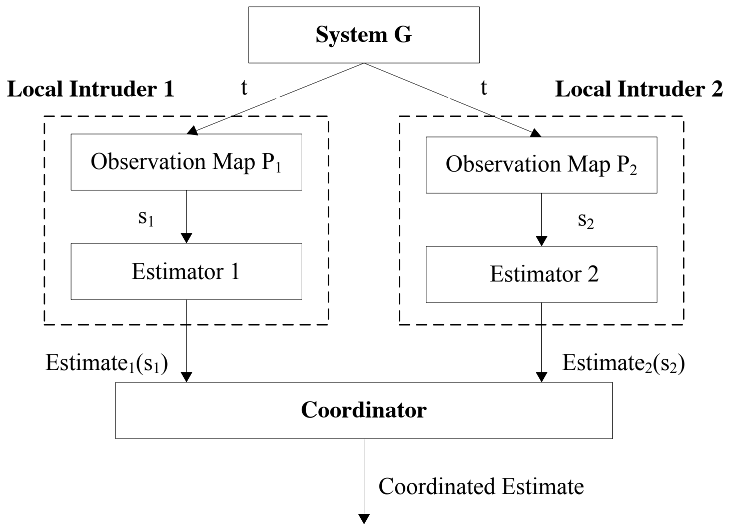


Figure 3.6: The coordinated architecture.

3.4.2 Joint-Initial-State Opacity (J-ISO)

We start by defining the notion of *joint initial-state opacity* in the coordinated architecture in Figure 3.6.

Definition III.26 (Joint-Initial-State Opacity (J-ISO)). Given G, projection maps P_1 and P_2, set of secret states $X_S \subseteq X_0$, and set of non-secret states $X_{NS} \subseteq X_0$, G is jointly initial-state opaque under the coordinated architecture if $\forall i \in X_S$ and $\forall t \in \mathcal{L}(G, i)$, $\exists j \in X_{NS}$, $\exists t_1 \in \mathcal{L}(G, j)$, $\exists t_2 \in \mathcal{L}(G, j)$ such that $P_1(t) = P_1(t_1) = s_1$, and $P_2(t) = P_2(t_2) = s_2$.

System G is jointly initial-state opaque (J-ISO) if for every string t from a secret state i in X_S, there are strings t_1 and t_2 from a common non-secret state j such that t_1 and t_2 are observationally equivalent to t for intruders 1 and 2, respectively. The common non-secret initial state j ensures that the coordinated estimate, formed by the intersection of the two local estimates, contains a non-secret state whenever the real initial state is a secret state. Note that if a given system is jointly initial-state opaque, it must be initial-state opaque for each local intruder. However, the reverse is not true in general.

We use the G_R-based ISE to model local intruders, where intruder k is represented by $Obs_k(G_R, X), k = 1, 2$. In addition, for analysis purposes, we consider $Obs(G_R, X)$ which models the *system intruder* whose observable event set is E_o; it is not a real intruder and its role will be explained later.

To verify J-ISO, we introduce a *test automaton* to capture the coordinated estimate. The test automaton is defined as $G^{ISO}_{test} := (Q, E_o, f^{ISO}_{test}, q_0)$. The state space is $Q \subseteq X_{obs,R,1} \times X_{obs,R,2} \times X_{obs,R} \times X_{obs,R,1\cap 2}$, where $X_{obs,R,1}, X_{obs,R,2}$, and $X_{obs,R}$ are the state spaces of the G_R-based ISE for intruder 1, intruder 2 and the system intruder, and where $X_{obs,R,1\cap 2} := \{y \in 2^X : (\exists x_k \in X_{obs,R,k}, k = 1, 2)[y = x_1 \cap x_2]\}$. A state in Q is denoted as $q = (q_1; q_2; q_s; q_c)$ and the initial state of G^{ISO}_{test} is $q_0 = (q_{10}; q_{20}; q_{s0}; q_{10} \cap q_{20}) = (X; X; X; X)$. The transition function f^{ISO}_{test} is defined as follows:

$$f^{ISO}_{test}((q_1; q_2; q_s; q_c), e) =$$
$$\begin{cases} (f_{obs,1,R}(q_1, e);\ q_2;\ f_{obs,R}(q_s, e);\ f_{obs,1,R}(q_1, e) \cap q_2), \text{ if } e \in E_{o,1} \setminus E_{o,2} \\ (q_1;\ f_{obs,2,R}(q_2, e);\ f_{obs,R}(q_s, e);\ q_1 \cap f_{obs,2,R}(q_2, e)), \text{ if } e \in E_{o,2} \setminus E_{o,1} \\ (f_{obs,1,R}(q_1, e);\ f_{obs,2,R}(q_2, e);\ f_{obs,R}(q_s, e);\ f_{obs,1,R}(q_1, e) \cap f_{obs,2,R}(q_2, e)) \\ , \text{ if } e \in E_{o,1} \cap E_{o,2} \end{cases}$$

where $e \in E_o$, $f_{obs,k,R}$ is the transition function of $Obs_k(G_R, X)$, and $f_{obs,R}$ is the transition function of $Obs(G_R, X)$. Note that the fourth state $q_c = q_1 \cap q_2$ does not affect the behavior of G^{ISO}_{test}, and the dynamics of G^{ISO}_{test} depend only on $(q_1; q_2, q_s)$ and are equivalent to that of $Obs_1(G_R, X) \parallel Obs_2(G_R, X) \parallel Obs(G_R, X)$. The additional parallel composition with $Obs(G_R, X)$ is to restrict the behavior of the test automaton within the system's observable behavior:

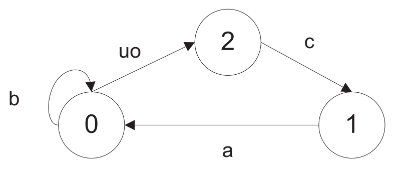
$$\begin{aligned}\mathcal{L}(G_{test}^{ISO}) :&= P_{o,1}^{-1}\big(\mathcal{L}[Obs_1(G_R, X)]\big) \cap P_{o,2}^{-1}\big(\mathcal{L}[Obs_2(G_R, X)]\big) \cap \mathcal{L}[Obs(G_R, X)] \\ &= \mathcal{L}[Obs(G_R, X)]\end{aligned}$$

where the inverse projection maps $P_{o,k}^{-1} : E_{o,k}^* \to 2^{E_o^*}$, for $k = 1, 2$, are with respect to E_o but not E.

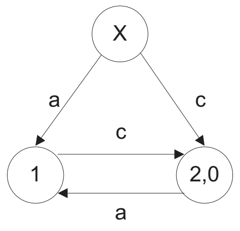
We use the following example to show how the system intruder, $Obs(G_R, X)$, restricts the observable behavior of G_{test}^{ISO}.

Example III.27. Consider the system in Figure 3.7(a) with $E_{o,1} = \{a, c\}$ and $E_{o,2} = \{b, c\}$. To verify J-ISO, we first build two G_R-based ISE to model intruders 1 and 2, as shown in Figures 3.7(b) and 3.7(c), respectively. Then, we want to synchronize the two local intruders and obtain the coordinated estimates. To synchronize the two local intruders, the first method that comes in mind is to parallel compose $Obs_1(G_R, X)$ and $Obs_2(G_R, X)$ and generate coordinated estimates that are the intersection of the two local estimates. Such an automaton is shown in Figure 3.8(a); however, it does not give the correct system behavior. This automaton generates strings outside of the system's observable behavior $P[\mathcal{L}(G_R, X)]$. For example, it generates string ab while the system does not generate its reversed string ba. To restrict the behavior of the test automaton within the system's observable behavior, we do an additional parallel composition with the system intruder $Obs(G_R, X)$ that is shown in Figure 3.7(d), and obtain the coordinated estimates in the fourth component by taking the intersection of the two local estimates. The resulting test automaton G_{test}^{ISO} is shown in Figure 3.8(b). One can see that it no longer includes string ab. In fact, the test automaton generates $\mathcal{L}[Obs(G_R, X)]$.

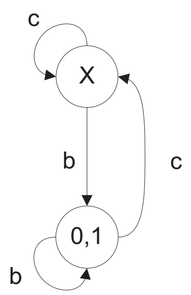
Let us denoted by $\hat{X}_{0,coor}(s)$ the coordinated initial state estimate for string $t \in$



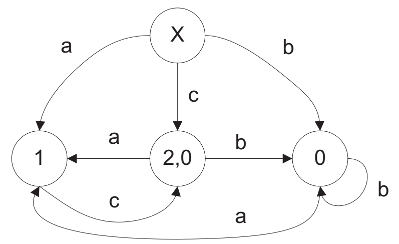
(a) The DES in Example III.27



(b) The G_R-ISE for intruder 1, $E_{o,1} = \{a, c\}$

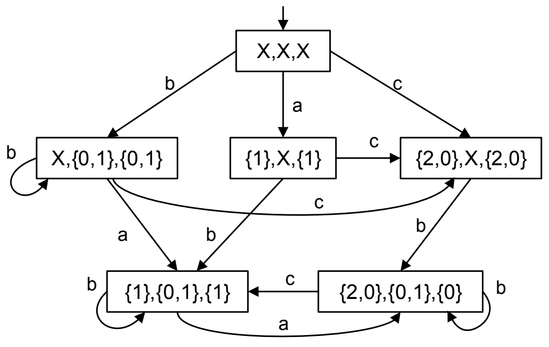


(c) The G_R-based ISE for intruder 2, $E_{o,2} = \{b, c\}$

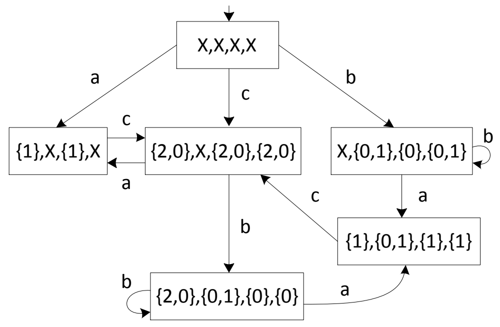


(d) The G_R-based ISE for the system intruder, $E_o = E_{o,1} \cup E_{o,2}$

Figure 3.7: The DES and its local and system G_R-based ISEs used in Example III.27.



(a) The incorrect G_{test}^{ISO}



(b) The correct G_{test}^{ISO}

Figure 3.8: The G_{test}^{ISO} in Example III.27

$\mathcal{L}(G, X)$ with $P(t) = s$. Lemma III.28 shows that $\hat{X}_{0,coor}(s)$ for every $s \in P[\mathcal{L}(G, X)]$ is captured by G_{test}^{ISO}. We will show later how to use G_{test}^{ISO} to verify the J-ISO property

Lemma III.28. *Given G, projection maps P_1 and P_2, the coordinated initial-state estimate for $s \in P[\mathcal{L}(G, X)]$ is captured by the state reached by s_R in G_{test}^{ISO}. Specifically, $\hat{X}_{0,coor}(s) = q_c$ where q_c is the fourth element in $q = (q_1; q_2; q_s; q_c) = f_{test}^{ISO}(q_0, s_R)$*

Proof. Pick any string $s \in P[\mathcal{L}(G, X)]$ with $P_1(s) = s_1$, and $P_2(s) = s_2$. By Theorem III.13, the initial state estimates of the local intruder 1 and 2 are $\hat{X}_{0,1}(s_1)= f_{obs,1,R}(X, s_{1R})$ and $\hat{X}_{0,2}(s_2)= f_{obs,2,R}(X, s_{2R})$. Thus, the coordinated initial state estimate is $\hat{X}_{0,coor}(s) = \hat{X}_{0,1}(s_1) \cap \hat{X}_{0,2}(s_2) = f_{obs,1,R}(X, s_{1R}) \cap f_{obs,2,R}(X, s_{2R})$. On the other hand, by construction, the state reached by s_R in G_{test}^{ISO} is $f_{test}^{ISO}(q_0, s_R) = (q_1; q_2; q_s; q_c)$, which is $(f_{obs,1,R}(X, s_{1R});$
$f_{obs,2,R}(X, s_{2R});\ f_{obs,R}(X, s_R);\ f_{obs,1,R}(X, s_{1R}) \cap f_{obs,2,R}(X, s_{2R}))$, where the fourth element q_c is $f_{obs,1,R}(X, s_{1R}) \cap f_{obs,2,R}(X, s_{2R}) = \hat{X}_{0,coor}(s)$. Therefore, the coordinated estimate $\hat{X}_{0,coor}(s)$ for $s \in P[\mathcal{L}(G, X)]$ is the fourth element of state $q = f_{test}^{ISO}(q_0, s_R)$. □

We now use G_{test}^{ISO} to verify J-ISO.

Theorem III.29. *G is jointly initial-state opaque if and only if $\forall q = (q_1; q_2; q_s; q_c)$ reachable in G_{test}^{ISO}, $q_c \cap X_S \neq \emptyset \Rightarrow q_c \cap X_{NS} \neq \emptyset$.*

Proof. G is J-ISO if and only if $\forall s \in P[\mathcal{L}(G, X)], \hat{X}_{0,coor}(s)$ always contains a non-secret state if it contains a secret state. To prove this Theorem, it is sufficient to prove that $\{\hat{X}_{0,coor}(s) : s \in P[\mathcal{L}(G, X)]\} = \{q_c : (\exists q = (q_1; q_2; q_s; q_c))[q \text{ is reachable in } G_{test}^{ISO}]\}$. By Lemma III.28, we know that for every $\hat{X}_{0,coor}(s)$ in the left-hand side, there is state q reached via s_R in the right-hand side. Since G_{test}^{ISO} is deterministic and $\mathcal{L}(G_{test}^{ISO}) = \mathcal{L}[Obs(G_R, X)] = Rev\big(P[\mathcal{L}(G, X)]\big)$, every reachable state of G_{test}^{ISO} in the right-hand side also corresponds to a valid $\hat{X}_{0,coor}(\cdot)$ in the left-hand side. Therefore, the above equality holds and J-ISO can be verified by examining the fourth element of all reachable states in G^{ISO}_{test}. □

Example III.30. Let us go back to Example III.27 and take the secret and non-secret state sets to be $X_S = \{0\}$ and $X_{NS} = X \setminus X_S$. The system is initial-state opaque to each local intruder because no state in $Obs_1(G_R, X)$ or $Obs_2(G_R, X)$ contains only the secret state 0. However, the system is not jointly initial-state opaque. As seen in Figure 3.8(b), by collaborating under the coordinated architecture, the team of intruders generate a coordinated estimate $\{0\}$ when $P(t) = s = bc$ ($s_R = cb$) has occurred.

We can extend the above verification procedure for J-ISO to the joint versions of CSO and IFO, by employing the respective estimator in the construction of the test automaton. These results are presented in the next two sections.

3.4.3 Joint-Current-State Opacity (J-CSO)

Definition III.31 (Joint-Current-State Opacity (J-CSO))**.** Given G, projection maps P_1, P_2, set of secret states $X_S \subseteq X$, and set of non-secret states $X_{NS} \subseteq X$. G is jointly current-state opaque under the coordinated architecture if $\forall i \in X_0$ and $\forall t \in \mathcal{L}(G, i)$ such that $\exists x \in X_S, x \in f(i, t)$, we have $\exists j_1, j_2 \in X_0$, $\exists t_1 \in \mathcal{L}(G, j_1)$, $\exists t_2 \in \mathcal{L}(G, j_2)$ such that (i) $\exists x' \in X_{NS}, x' \in f(j_1, t_1), x' \in f(j_2, t_2)$ and (ii) $P_1(t) = P_1(t_1) = s_1$, $P_2(t) = P_2(t_2) = s_2$.

System G is jointly-current-state opaque if for every string t that reaches a secret state x, there are other two strings t_1 and t_2 reaching a common non-secret state x' such that intruder 1 confuses t with t_1 and intruder 2 confuses t with t_2. Strings t_1 and t_2 need not start from the same initial state, but they have to reach a common non-secret state to ensure that the non-secret state exists in the coordinated estimate. Similarly to the case of J-ISO, to verify J-CSO, we build a test automaton

$G_{test}^{CSO} := (Q, E_o, f_{test}^{CSO}, q_0)$ where the standard observers are used to model intruders. Specifically, $Obs_k(G)$ models the behavior of local intruder $k, k = 1, 2$, and $Obs(G)$ models the system intruder who observes E_o. The system intruder is to confine the behavior of the test automaton. The state space is $Q \subseteq X_{obs,1} \times X_{obs,2} \times X_{obs} \times X_{obs,1\cap 2}$ where $X_{obs,1}, X_{obs,2}$, and X_{obs} are state spaces of the corresponding observers and $X_{obs,1\cap 2} = \{y \in 2^X : (\exists x_k \in X_{obs,k}, k = 1, 2)[y = x_1 \cap x_2]\}$. A state in Q is denoted by $q = (q_1; q_2; q_s; q_c)$, and the initial state is $q_0 = (X; X; X; X)$. The transition function f_{test}^{CSO} is defined as follows:

$$f_{test}^{CSO}((q_1; q_2; q_s; q_c), e) =$$
$$\begin{cases} (f_{obs,1}(q_1, e);\ q_2;\ f_{obs}(q_s, e);\ f_{obs,1}(q_1, e) \cap q_2), \text{ if } e \in E_{o,1} \setminus E_{o,2} \\ (q_1;\ f_{obs,2}(q_2, e);\ f_{obs}(q_s, e);\ q_1 \cap f_{obs,2}(q_2, e)), \text{ if } e \in E_{o,2} \setminus E_{o,1} \\ (f_{obs,1}(q_1, e);\ f_{obs,2}(q_2, e);\ f_{obs}(q_s, e);\ f_{obs,1}(q_1, e) \cap f_{obs,2}(q_2, e)), \text{ if } e \in E_{o,1} \cap E_{o,2} \end{cases}$$

where $f_{obs,k}$ is the transition function of $Obs_k(G)$, and f_{obs} is that of $Obs(G)$.

Theorem III.32. *G is jointly current-state opaque if and only if $\forall q = (q_1; q_2; q_s; q_c)$ reachable in G_{test}^{CSO}, $q_c \cap X_S \neq \emptyset \Rightarrow q_c \cap X_{NS} \neq \emptyset$.*

The proof of Theorem III.32 is similar to that of Theorem III.32 because of the similarity between G_{test}^{CSO} and G_{test}^{ISO}. For this reason, we omit the proof.

3.4.4 Joint-Initial-and-Final-State Opacity (J-IFO)

Definition III.33 (Joint-Initial-and-Final-State Opacity (J-IFO))**.** Given G, projection maps P_1 and P_2, set of secret pairs $X_{sp} \subseteq X_0 \times X$, and set of non-secret pairs $X_{nsp} \subseteq X_0 \times X$, G is jointly initial-and-final-state opaque under the coordinated architecture if $\forall (x_0, x_f) \in X_{sp}$ and $\forall t \in \mathcal{L}(G, x_0)$ such that $x_f \in f(x_0, t)$, we have $\exists (x_0', x_f') \in X_{nsp}, \exists t_1, t_2 \in \mathcal{L}(G, x_0')$ such that (i) $\exists x_f' \in X_{NS}, x_f' \in f(x_0', t_1), x_f' \in f(x_0', t_2)$, and (ii) $P_1(t) = P_1(t_1) = s_1$, $P_2(t) = P_2(t_2) = s_2$.

G is jointly-initial-and-final-state opaque if for every string t that corresponds to a secret state pair, there are other two strings t_1 and t_2 corresponding to a common non-secret state pair $(x'_0, x'_f) \in X_{nsp}$ such that intruder 1 confuses t_1 with t and intruder 2 confuses t_2 with t. The common non-secret state pair $(x'_0, x'_f) \in X_{nsp}$ ensures that a non-secret state pair exists in the coordinated estimate.

To verify J-IFO, we use a test automaton $G^{IFO}_{test} := (Q, E_o, f^{IFO}_{test}, q_0)$ where trellis-based ISE are used to model local intruders and the system intruder. The state space is $Q \subseteq M_1 \times M_2 \times M \times M_{1\cap 2}$, where M_1, M_2 and M are the state spaces of corresponding trellis-based ISE, and $M_{1\cap 2} := \{y \in 2^{X^2} : (\exists m_k \in M_k, k = 1,2)[y = m_1 \cap m_2]\}$. A state in Q is denoted as $q := (q_1; q_2; q_s; q_c)$. The initial state of G^{IFO}_{test} is $q_0 = (q_{10}; q_{20}; q_{s0}; q_{10} \cap q_{20})$ where $q_{k0} = \{(i,i) : i \in X)\} \cup \{(i,j) \in X^2 : (\exists t \in E^*_{uo,k})[j \in f(i,t)]\}$ and $q_{s0} = \{(i,i) : i \in X)\} \cup \{(i,j) \in X^2 : (\exists t \in E^*_{uo})[j \in f(i,t)]\}$. The transition function f^{IFO}_{test} is defined as follows:

$$f^{IFO}_{test}((q_1; q_2; q_s; q_c), e) =$$
$$\begin{cases} (f_{tre,1}(q_1, e);\ q_2;\ f_{tre}(q_c, e);\ f_{tre,1}(q_1, e) \cap q_2), \text{ if } e \in E_{o,1} \setminus E_{o,2} \\ (q_1;\ f_{tre,2}(q_2, e);\ f_{tre}(q_c, e);\ q_1 \cap f_{tre,2}(q_2, e)), \text{ if } e \in E_{o,2} \setminus E_{o,1} \\ (f_{tre,1}(q_1, e);\ f_{tre,2}(q_2, e);\ f_{tre}(q_c, e);\ f_{tre,1}(q_1, e) \cap f_{tre,2}(q_2, e)), \text{ if } e \in E_{o,1} \cap E_{o,2} \end{cases}$$

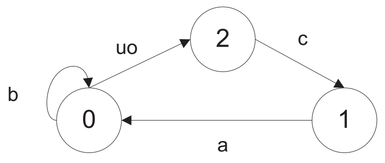
where $f_{tre,k}$ is the transition function of trellis ISE k, and f_{tre} is the transition function of the system trellis-based ISE.

Example III.34. Consider J-IFO using system G in Figure 3.9(a) with $X_{sp} = \{(0,2)\}$, $X_{nsp} = X^2 \setminus X_{sp}$, $E_{o,1} = \{a, c\}$, and $E_{o,2} = \{b, c\}$. We construct trellis-based ISE to model intruders 1 and 2 and the system intruder, as shown in Figure 3.9(b), 3.9(c), 3.9(d), respectively. One can see that the system is IFO to both local intruders. To verify J-IFO, we build tester G^{IFO}_{test}, as shown in Figure 3.10, by parallel composing the three trellises and adding the intersection of the two local estimates as the fourth element. The system is J-IFO because no states of G_{test}^{IFO} contains the secret state pair $(0, 2)$ alone.

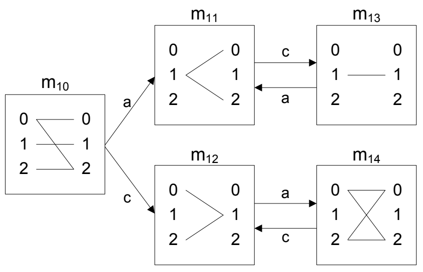
3.5 Discussion: Fast Intruders

So far, we have considered unobservable reach when we build the intruders' estimates. This means that our intruders, upon observing the latest observable event e_o, do not know whether other unobservable events have occurred after e_o. Hence, when constructing estimates, they take into account all the unobservable tails after e_o.

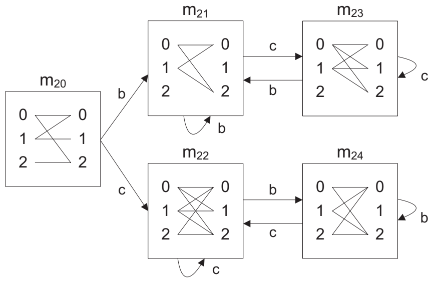
On the other hand, the authors in [11] have considered "fast intruders", which are intruders that react faster than the execution of system. Specifically, their intruders, upon observing an observable event, can update estimates before the occurrence of



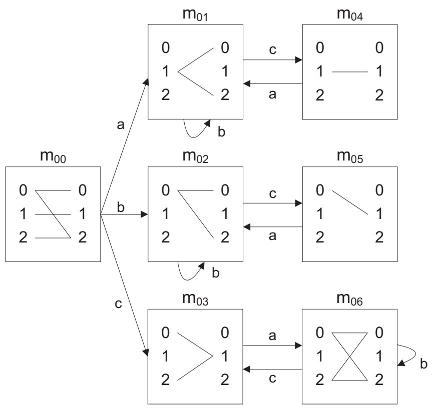
(a) The DES in Example III.34



(b) The trellis-based ISE for intruder 1, $E_{o,1} = \{a, c\}$



(c) The trellis-based ISE for intruder 2, $E_{o,2} = \{b, c\}$



(d) The trellis-based ISE for the system intruder, $E_o = \{a, b, c\}$

Figure 3.9: The DES and trellises used in Example III.34.

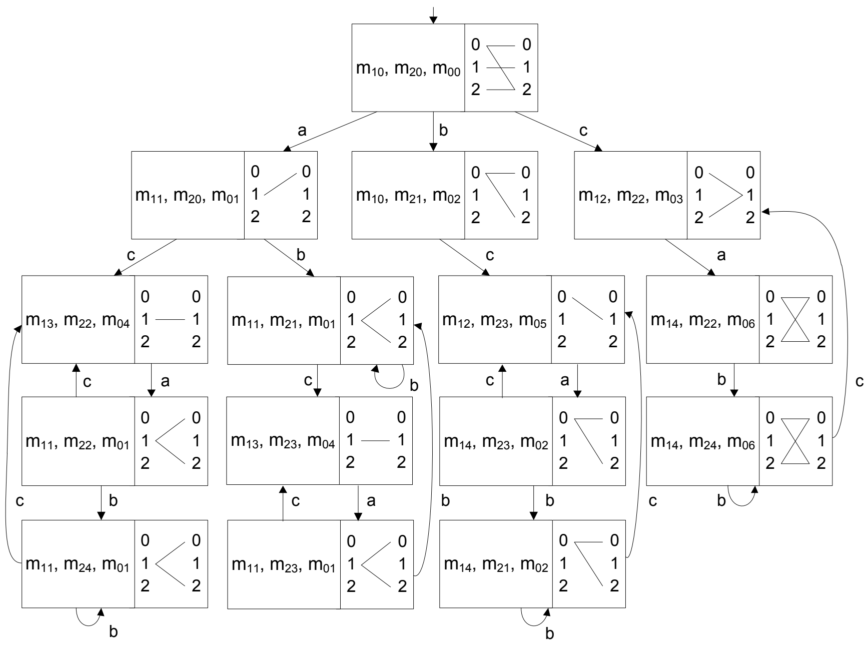


Figure 3.10: The tester G_{test}^{IFO} of the system in Example III.34

the next unobservable event. Hence, their intruders' estimates do not consider strings with unobservable tails. We now discuss how the assumption of "fast intruders" affects our results.

We first consider opacity notions in the centralized case. When verifying ISO using the G_R-based ISE, we consider unobservable reach when constructing $Obs(G_R, X)$. Because all strings in $Obs(G_R, X)$ are reversed, this unobservable reach is in fact the unobservable beginnings but not unobservable tails in the forward order. That is, the G_R-based ISE indeed model fast intruders. Note that an intruder always considers unobservable beginnings no matter whether it is a fast intruder or not. In fact, the G_R-based ISE also applies to all intruders, fast or not. This is because the unobservable reach changes only the final state estimates but not the initial state estimates. It simply extends already-considered strings with unobservable tails. Therefore, verification of ISO using the G_R-based ISE is not affected by the assumption of fast intruders. When using observers to verify CSO, we can choose to build the observer without the unobservable reach to model fast intruders. The fast intruder generates a more accurate estimate right after the system makes an observable transition. However, as the system may invisibly move, the estimate of the fast intruder is only temporarily correct. Similarly, we could use the trellis-based ISE where the state mappings include strings with no unobservable tails and the initial state is $\{(i, i) : i \in X)\}$ to model fast intruders in the IFO problem. The final state estimate of the trellis-ISE is also only temporarily correct. System designers may choose whether or not to use the unobservable reach based on the architecture of the application.

The assumption of fast intruders affects the current (final) state estimates but not the initial state estimates. Thus, for opacity notions in the coordinated architecture, this assumption will only affect our results for J-CSO and J-IFO. In both cases, each fast local intruder's estimate is only temporarily correct and each of them updates its estimate upon its individual set of observable events. Therefore, the coordinator may obtain an empty intersection when one intruder, say intruder 1, just updates its estimate upon an event in $E_{o,1} \setminus E_{o,2}$ and the estimate of intruder 2 is incorrect. If we want to take advantage of fast intruders and avoid the problem of empty intersections, we could design a communication protocol similar to that in [16] where intruder k, $k = 1, 2$, sends two estimates, $\hat{X}_k^f(s)$ and $\hat{X}_k(s)$, where $\hat{X}_k^f(s)$ is the fast estimate without the unobservable reach and $\hat{X}_k(s)$ is the estimate with the unobservable reach corresponding to $E_{uo,k}$. If the coordinator receives an estimate from only one intruder, say intruder 1, it forms the coordinated estimate $\hat{X}_1^f(s) \cap \hat{X}_2(s)$; if it receives estimates from both intruders, the coordinated estimate is $\hat{X}_1^f(s) \cap \hat{X}_2^f(s)$.

3.6 Conclusion

We have presented several new results regarding the notion of opacity in DES in the context of centralized and coordinated architectures. Four types of opacity notions have been investigated: language-based opacity, initial-state opacity, current-state opacity, and initial-and-final-state opacity; the latter one was introduced in Chapter

II to capture situations where the secret simultaneously involves the initial and final states of the system. We have also developed a set of transformation algorithms between the four notions of opacity and showed that every pair of opacity notions are equivalent through a polynomial reduction. These results unify the treatment of opacity in DES. To verify different notions of opacity, we have reviewed existing methods and proposed new algorithms. Finally, we have formulated three new notions of joint opacity in the context of a coordinated architecture where a set of intruders work as a team to infer the secret. Verification algorithms have been proposed for each notion of joint opacity, leveraging the centralized tests.

CHAPTER IV

Opacity Enforcement Using Event Insertion

4.1 Introduction

In the previous chapter, we have discussed the algorithms for verifying the four notions of opacity. When a given opacity notion fails to hold, the ensuing question is: How can we enforce the secret to be opaque? In this chapter, we consider the problem of enforcing opacity and develop a new opacity enforcement mechanism using insertion functions at run-time. As shown in Figure 4.1, an insertion function is a monitoring interface placed at the output of the system. It receives run-time output from the system, inserts an additional observable string if necessary in order to prevent the system from revealing the secret, and outputs the modified behavior.

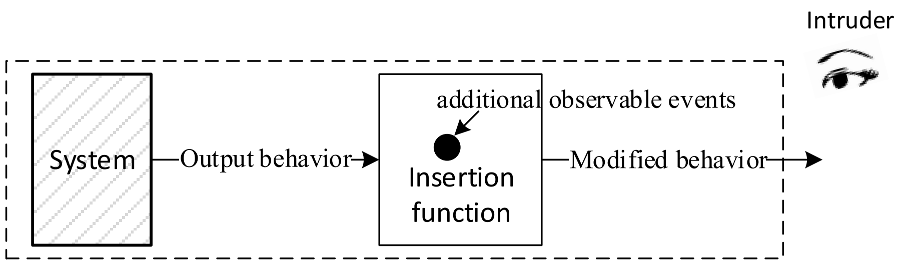


Figure 4.1: The insertion mechanism

We consider insertion functions that cannot interfere with the outputting of events from the system; i.e., they must allow every system's output behavior. Also, we require every modified output from the insertion function to be observationally equivalent to some original non-secret behavior. This is to ensure that the intruder cannot learn the existence of the insertion function from observing the modified output. Specifically, the intruder is assumed to have no knowledge of the insertion function at the outset. But with the knowledge of the system's structure, it may learn the existence of the insertion function from observing the modified output if, for instance, the insertion function inserts random events. Hence, the insertion function should only output behavior that could occur in the original system. We characterize the above two requirements as the *i-enforceability* property. Given an opacity notion, it is *i-enforceable* if there exists an i-enforcing insertion function.

The main focus of this chapter is to synthesize an i-enforcing insertion function when the given secret of the system is not opaque. We begin with verifying if the given opacity notion is i-enforceable; i.e., if there exists an i-enforcing insertion function. This is done by constructing a structure called the "All Insertion Structure" (AIS) that enumerates in a finite structure all i-enforcing insertion functions. If opacity is i-enforceable, we then use the AIS to *synthesize* an i-enforcing insertion function. All these algorithms are general enough so that they apply to four opacity properties: current-state opacity, initial-state opacity, language-based opacity, and initial-and-final-state opacity.

This chapter is organized as follows. Section 4.2 presents the relevant definitions. Section 4.3 formally defines insertion functions and the i-enforcing property. In Section 4.4, we present our algorithm for verifying if a given insertion function is i-enforcing. In Section 4.5, we show the construction of the All Insertion Structure (AIS) and verify i-enforceability of a given opacity notion using the AIS. Section 4.6 presents the synthesis of an i-enforcing insertion function. The complexity of the AIS is analyzed in Section 4.7. Sections 4.8 proposes a more efficient algorithm for constructing the AIS. Section 4.9 discusses opacity enforcement by insertion in the context of opaque communications. Section 4.10 discusses how intruder's knowledge of the insertion function affects our results. Finally, Section 4.11 concludes this chapter. Most of the results in this chapter also appear in [60].

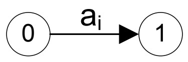
4.1.1 Related Work

Many prior studies have considered the enforcement of opacity notions. The works in [19, 46, 56] have designed the minimally-restrictive opacity-enforcing supervisory controller based on the supervisory control theory of DES. With this approach, the system's behavior is restricted such that a behavior is disabled by feedback control if it is going to reveal the secret. Our approach differs from this approach in that insertion functions are not allowed to interfere with the system and must allow the system to execute its full behavior.

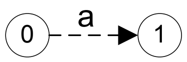
Another approach to enforce opacity notions is to use a dynamic observer, which dynamically modifies the observability of every system event [11]. Similar to an i-enforcing insertion function, a dynamic observer does not interfere with the system output and allows the full system behavior. However, a dynamic observer may create "new" observed strings that would not be seen in the original system (under a static observable projection), as it dynamically erases information that was to be output.

The authors in [23] have also proposed a run-time opacity enforcement mechanism that allows the full system behavior. This work employs delays when outputting executions in order to enforce K-step opacity. However, this method applies only to secrets for which time duration is of concern.

Other works in the computer science literature have also used insertion functions to enforce security properties; see e.g., [32, 49]. However, the class of security policies considered does not include opacity. To the best of our knowledge, our work is the first to address opacity enforcement using insertion functions.



(a) Inserted event a denoted by the subscript i



(b) Inserted event a denoted by a dashed line

Figure 4.2: Two representations of the insertion label

4.2 Preliminaries

4.2.1 Relevant Definitions and Operations

Inserted Event Set E_i

In our enforcement mechanism, the insertion function can insert any event in E_o. Such an inserted event looks identical to a system observable event. However, for the purpose of discussion, we want to clearly distinguish between inserted events and observable events. Thus, we define a set of inserted events E_i, where an insertion label is attached to each event, resulting in $E_i = \{e_i : e \in E_o\}$. The insertion label is presented by either the subscript i of event labels, as in Figure 4.2(a), or by the dashed feature of transition arrows; as in Figure 4.2(b). We will choose to present the label by one of these methods, depending on the context.

Projection P_{und} and Mask $\mathcal{M}_i$

Projection P_{und} and mask $\mathcal{M}_i$ are defined to analyze strings consisting of events in $E_i \cup E_o$. P_{und} is a natural projection that treats dashed transitions as unobservable. The subscript *und* is short for "undashed". Given an event, P_{und} outputs the empty string if the event is inserted and outputs the original event if the event is observable: $P_{und}(e_i) = \varepsilon, e_i \in E_i$ and $P_{und}(e) = e, e \in E_o$. On the other hand, $\mathcal{M}_i$ is a mask that treats inserted events and observable events as indistinguishable. Given an event, $\mathcal{M}_i$ removes the subscript i if it exists and outputs the same event otherwise:

$\mathcal{M}_i(e_i) = e, e_i \in E_i$ and $\mathcal{M}_i(e) = e, e \in E_o$.

Projection P_{io}

Projection P_{oi} is a natural projection that maps from $E \cup E_i$ to $E_o \cup E_i$. Formally, given $e \in E \cup E_i$, $P_{oi}(e) = e$ if $e \in E_o \cup E_i$ and $P_{oi}(e) = \varepsilon$ if $e \in E_{uo} \cup \{\varepsilon\}$. The subscript oi means "observable and inserted events".

Dashed Parallel Composition

The *dashed parallel composition* is a special synchronization operator denoted as $||_d$. This composition synchronizes two types of automata: one with only solid transitions (e.g., the automaton in Figure 4.6(a)), and one with both solid and dashed transitions (e.g., the automaton in Figure 4.6(b)). In dashed parallel composition, the transitions of the two automata are synchronized on common event labels, like in the standard parallel composition. However, given a common event, it is represented by a dashed transition if the corresponding transition in the *second* automaton is a dashed one, and by a solid transition otherwise. For private events, the solid/dashed feature of the transitions is preserved. The resulting dashed parallel composition of the automata in Figures 4.6(a) and 4.6(b) is shown in Figure 4.6(c).

Input Parallel Composition

The input parallel composition, denoted as $||_{i/p}$, is an operation between a standard automaton $G = (X_1, E_1, f_1, X_{0,1})$ and a deterministic Mealy automaton $M = (X_2, E_{i,2}, E_{o,2}, f_2, q_2, x_{0,2})$. The input parallel composition of G and M is a nondeterministic Mealy automaton $A = G||_{i/p}M = (X_1 \times X_2, E_1, E_{o,2}, f, q, X_{0,1} \times \{x_{0,2}\})$

where the state transition function f and the output function q are defined as:

$$f((x_1,x_2),e) = \begin{cases} \{(x_1', f_2(x_2,e)) : x_1' \in f_1(x_1,e)\} \\ \text{ if } f_1(x_1,e) \text{ and } f_2(x_2,e) \text{ are defined} \\ \{(x_1', x_2) : x_1' \in f_1(x_1,e)\} \\ \text{if } f_1(x_1,e) \text{ is defined but} f_2(x_2,e) \text{ is undefined} \end{cases}$$

$$q((x_1,x_2),e) = \begin{cases} q_2(x_2,e), \text{ if } f_1(x_1,e) \text{ and } f_2(x_2,e) \text{ are defined} \\ e, \text{ if } f_1(x_1,e) \text{ is defined but } f_2(x_2,e) \text{ is undefined} \end{cases}$$

That is, $(x_1,x_2) \xrightarrow{e/t} (x_1',x_2')$ where $q_2(x_2,e) = t$ if $f_1(x_1,e)$ and $f_2(x_2,e)$ are both defined; $(x_1,x_2) \xrightarrow{e/e} (x_1',x_2)$ if only $f_1(x_1,e)$ is defined.

The definition of f shows that only events in E_1 can trigger transitions in $G||_{i/p}M$. In fact, the two automata are synchronized in a master/slave manner where the Mealy automaton passively reacts to the events of the standard automaton. Such synchronization is suitable for our insertion mechanism because the insertion function's response is driven by the system's output behavior. Also, because the insertion function reacts only to observable behavior, we can always let $E_2 = E_o \subseteq E_1 = E$ in our examples without loss of generality.

4.2.2 Safe Language, Safe Strings, and Safe Estimates

In this chapter, we consider opacity enforcement for systems that are running *online*. Once the secret is revealed, opacity is violated and cannot be recovered. Given G, the *safe language* is the "largest" sublanguage of $P[\mathcal{L}(G)]$ that *never* reveals the occurrence of the secret. A *safe string* is a string in the safe language. A *safe estimate* is a state of the "forward" state estimator that corresponds to a safe string.[1]

Recall from Chapter III that we can verify each of the four notions considered in this thesis by mapping it to LBO and checking if $P[\mathcal{L}(G)] \subseteq P(L_{NS})$. The safe language, denoted by L_{safe}, is the *supremal prefix-closed sublanguage* of $P(L_{NS})$ and is characterized in Equation 4.1 using the result from [31].

$$L_{safe} = P[\mathcal{L}(G)] \setminus (P[\mathcal{L}(G)] \setminus P(L_{NS}))\, E_o^* \tag{4.1}$$

Opacity holds if the system only outputs strings in L_{safe}; i.e., $P[\mathcal{L}(G)] \subseteq L_{safe}$. Notice that L_{safe} is regular if $P(L_{NS})$ is regular. We call string $s \in P[\mathcal{L}(G)]$ safe if $s \in L_{safe}$ and unsafe otherwise.

We can also verify opacity by building the corresponding forward state estimator and checking if any estimate contains only the secret information (specifically, current states, initial states, or initial-and-final-state pairs). A forward state estimator is an automaton where a state reached by string $s \in P[\mathcal{L}(G)]$ is the intruder's [current-state; initial-state; initial-and-final-state] *estimate* when the intruder observes s. Specifically, CSO and LBO can be verified by the standard *observer automaton* defined in Section 2.5.2 of [10]; ISO and IFO can be verified by the trellis-based Initial-State Estimator (ISE) introduced in [43]. For simplicity, we will call a forward state estimator an *estimator* and denote it by $\mathcal{E}$ hereafter.

We can determine if a given observed string is safe or unsafe by examining the estimate it reaches in $\mathcal{E}$. Specifically, string s is safe if no prefix of s reaches an estimate that contains only the secret information in $\mathcal{E}$, and is unsafe otherwise (recall that L_{safe} is prefix-closed). We call an estimate safe if it is reached by a safe string, and unsafe otherwise. It is clear that an estimate containing only the secret information is unsafe. However, an estimate containing non-secret information is still unsafe if

[1]This "forward" state estimator is in contrast to the G_R-based initial state estimator we proposed in [59], which is the observer of the reversed automaton of G.

the corresponding observed string has an unsafe prefix.

Consider an opacity notion. We now build from $\mathcal{E}$ an automaton that generates L_{safe}. This automaton, called the desired estimator and denote by $\mathcal{E}^d$, is built by deleting all estimates in $\mathcal{E}$ that contain only the secret information and taking the accessible part. This is in fact the implementation using automata of the language expression of L_{safe} in Equation (4.1). Hence, we have the following result:

Proposition IV.1. The desired estimator $\mathcal{E}^d$ generates L_{safe}; i.e., $\mathcal{L}(\mathcal{E}^d) = L_{safe}$.

Our results in the remainder of this chapter are developed based on L_{safe} and $\mathcal{E}$.

4.3 Enforcement of Opacity Using Insertion Functions

An insertion function is a special monitoring interface that not only monitors the system but also inserts additional events to the system output when necessary. As shown in Figure 4.1, the insertion function takes an observed *event* from the system, possibly inserts extra observable events, and outputs the resulting *string. In terms of the intruder, the extra inserted events are indistinguishable from genuine observable events.* That is, the intruder does not recognize the virtual insertion label. Therefore, the intruder, observing at the output of the insertion function, cannot tell if the observed string includes inserted events or not. Given a modified string $s \in (E_o \cup E_i)^*$ from the insertion function, the intruder observes s under mask $\mathcal{M}_i$. We can also reconstruct the genuine string by applying projection P_{und} to s. Notice that the intruder is assumed to have no knowledge of the insertion function at the outset. Our goal is to design an insertion function such that it protects the secret behavior and the intruder can never learn the existence of the insertion function based on its knowledge of the system structure and the observation of the modified output. Such a property will be characterized as "i-enforceability" in Section 4.3.2. We will also discuss in Section 4.10 how intruder's knowledge of the insertion function affects our results.

4.3.1 Insertion Functions and Insertion Automata

The insertion enforcement mechanism depends on the design of insertion functions. We define an insertion function as a (potentially partial) function $f_I : E_o^* \times E_o \rightarrow E_i^* E_o$ which outputs a string with insertions, based on the system's past and current observed behavior. More precisely, given that the system has executed string t where $P(t) = s$ and the current observed event is $e_o \in E_o$, the insertion function is defined such that $f_I(s, e_o) = s_I e_o$ if string $s_I \in E_i^*$ is inserted before e_o. Hereafter, we will use (s, e_o) to denote the system's observed behavior that defines the output of f_I. In the following, we assume that no s_I is of unbounded length.

The function f_I defines the instantaneous insertion for every (s, e_o). To determine the complete modified string from the insertion function, we define an equivalent string-based insertion function f_I^{str} from f_I: $f_I^{str}(\varepsilon) = \varepsilon$ and $f_I^{str}(s_n) = f_I(\varepsilon, e_1) f_I(e_1, e_2) \cdots f_I(e_1 e_2 \ldots e_{n-1}, e_n)$ where $s_n = e_1 e_2 \ldots e_n \in E_o^*$. Given G, the modified language output from the insertion function is $f_I^{str}(P[\mathcal{L}(G)]) = \{\tilde{s} \in (E_i^* E_o)^* : \tilde{s} = f_I^{str}(s) \wedge s \in P[\mathcal{L}(G)]\}$ and is denoted hereafter by L_{out}.

We encode a given insertion function f_I as a (possibly infinite state) Mealy automaton and call it insertion automaton, denoted by $IA = (X_{ia}, E_o, E_i^* E_o, f_{ia}, q_{ia}, x_{0,ia})$. Specifically, the state set is X_{ia}, the input set is E_o, the output set is the set of *strings* in $E_i^* E_o$, the transition function f_{ia} defines the dynamics of IA, the output function q_{ia} is defined such that $q_{ia}(x, e_o) = s_I e_o$ where $f_{ia}(x_{0,ia}, s) = x$, if $f_I(s, e_o) = s_I e_o$, and $x_{0,ia}$ is the initial state. In Figure 4.3, we show an insertion automaton that inserts a_i before the first output event if that event is b.

In general, an insertion automaton can have an infinite set of states. We call an insertion function "finite" if it can be encoded as a finite-state insertion automaton.

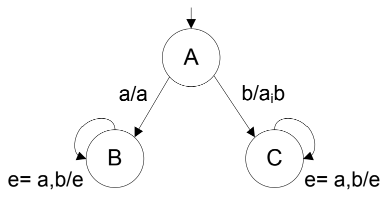


Figure 4.3: An example of insertion automata

4.3.2 I-Enforcing Insertion Functions

For an insertion function to enforce opacity in our insertion mechanism, it needs to satisfy a property called *i-enforcing*. I-enforceability characterizes an insertion function's ability to output strings that *always* look like a non-secret behavior under $\mathcal{M}_i$, without ever interacting with the system. Specifically, i-enforceability holds if both *safety* and *admissibility* hold. An insertion function is *i-enforcing* if it is safe and admissible.

Safety property defines the output from the insertion function, i.e., L_{out}. It requires every behavior after insertion to be within the safe language. If the system is monitored with a safe insertion function, no output reveals the secret.

Definition IV.2 (Safety)**.** Given G, secret language L_S and non-secret language L_{NS}, an insertion function is safe if the modified language under mask $\mathcal{M}_i$ is within the safe language; that is, $\mathcal{M}_i\left(L_{out}\right) \subseteq L_{safe}$.

Admissibility property defines the input to the insertion function. It is an interface feature that requires the insertion function to take every system's observable behavior as a valid input. For every behavior in $P[\mathcal{L}(G)]$, an admissible insertion function must respond to the behavior with an insertion.

Definition IV.3 (Admissibility)**.** Given G, an insertion function is admissible if it inserts (possibly ε) on every observable behavior from G. That is, $f_I(s, e_o)$ is defined for all $se_o \in P[\mathcal{L}(G)]$.

We now combine the two properties and define i-enforceability:

Definition IV.4 (I-Enforceability)**.** Given G, an insertion function is called *i-enforcing* if it is both safe and admissible. Moreover, the considered opacity notion is called *i-enforceable* if there exists an i-enforcing insertion function.

Safety guarantees that every modified behavior from f_I looks like a non-secret string from the original system. If an unsafe string is output by the system but rendered safe by f_I, the intruder will believe that a non-secret string has occurred. On the other hand, admissibility guarantees that f_I does not interact with the system (e.g., by disabling events). Hence, having both properties, our i-enforcing insertion function can always map the system output to a non-secret behavior no matter what the system outputs. Thus, the secret will never be revealed and opacity is enforced by the insertion function.

4.4 Verifying I-enforceability of An Insertion Function

Given an insertion function, it is desirable to know if it is i-enforcing. In this section, we propose an algorithm that verifies the i-enforcing property of the given insertion function. The algorithm is based on the construction of an equivalent automaton called $\tilde{G}$ that captures the behavior of the modified system. If the behavior of this equivalent automaton is within L_{safe}, then the insertion function is i-enforcing.

We will prove in Section 4.6 that if an i-enforcing insertion function exists, then a finite i-enforcing one exists. Thus, with no loss of generality, it is assumed in this section that the insertion function is specified in terms of a finite state insertion automaton. Given G and f_I, we check if f_I is i-enforcing by following Algorithm 1. To facilitate presentation, we denote the input language of IA by $\mathcal{L}_i(IA)$.

In Algorithm 1, step 1 checks the admissibility property of f_I. If the input language of IA contains $P[\mathcal{L}(G)]$, then f_I responds to all possible system outputs and is admissible. Otherwise, f_I is not admissible and the algorithm returns "not i-enforcing".

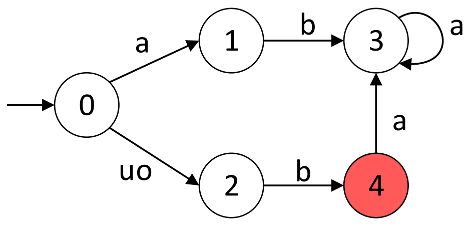
Algorithm 1: Verify i-enforceability of f_I

```
    input  : $G$, $f_I$ encoded as $IA$, and deterministic $H$ s.t. $\mathcal{L}(H) = L_{safe}$
    output: I-ENFORCING or NOT I-ENFORCING
1   if $P[\mathcal{L}(G)] \subseteq \mathcal{L}_i(IA)$ then
        Go to Step 2
    else
        Return NOT I-ENFORCING
2   Build Mealy automaton $M_i = G||_{i/p}IA$
3   Remove input labels in $M_i$ and extract automaton $G_i$
4   Extend $G_i$ to $\tilde{G}$ by recursively introducing new states:
    for $x, y \in X_{G_i}$ (the state space of $G_i$) s.t. $x \xrightarrow{t=et'} y$ where $e \in E_i$, $t' \in E_i^* E_o$ do
        introduce $x_{ins}$ to $X_{G_i}$ and redefine $x \xrightarrow{e} x_{ins} \xrightarrow{t'} y$
5   if $\mathcal{M}_i\left(P_{oi}[\mathcal{L}(\tilde{G})]\right) \subseteq L_{safe}$ then
        Return I-ENFORCING
    else
        Return NOT I-ENFORCING
```

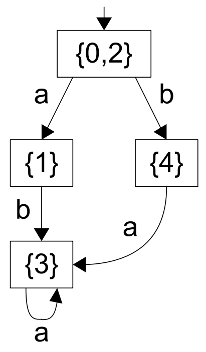
When f_I is admissible, we build $\tilde{G}$ in step 2-4 and check in step 5 if f_I is also safe. $\tilde{G}$ is an equivalent automaton that represents the modified system. The projected language of $\tilde{G}$ is the modified output behavior from f_I. If $\mathcal{M}_i\left(P_{oi}[\mathcal{L}(\tilde{G})]\right) \subseteq L_{safe}$, where $\mathcal{M}_i$ removes the virtual insertion label that is only for analysis and not seen by the intruder, then the intruder will never observe secret behaviors and thereby f_I will be safe. If f_I is admissible and safe, the algorithm returns "i-enforcing". Otherwise, the algorithm returns "not i-enforcing". Notice that an i-enforcing f_I maps every unsafe string to a safe string. When there is an unsafe string, $\tilde{G}$ always becomes nondeterministic under the effect of $\mathcal{M}_i \circ P_{oi}$ (more precisely, under the automaton implementation of the language map $\mathcal{M}_i \circ P_{oi}$).

Example IV.5. Consider system G in Figure 4.4(a) where $E_o = \{a, b\}$ and $X_S = \{4\}$. We build the estimator $\mathcal{E}$ in Figure 4.4(b) to verify current-state opacity (CSO). The system is not CSO because estimate $\{4\}$ contains only the secret state. To enforce opacity, we propose to use the f_I encoded in Figure 4.3 and verify if it is i-enforcing by following Algorithm 1. The input language of f_I is $(a + b)^*$, which contains

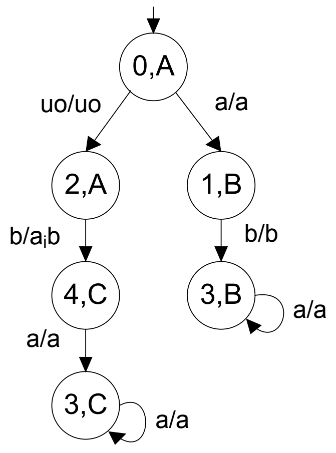
$P[\mathcal{L}(G)]$. Thus, f_I is admissible. Then, we construct the equivalent automaton $\tilde{G}$ to check if f_I is also safe. The intermediate Mealy automaton M_i is shown in Figure 4.4(c). In M_i, the output label of the transition from $(2, A)$ to $(4, C)$ is a string $a_i b$ instead of an event, meaning that f_I inserts a_i when the system moves from state 2 to state 4. Then by removing input labels in M_i and introducing dummy state Ins, we build the equivalent automaton $\tilde{G}$ in Figure 4.4(d). It can be proven that $\mathcal{M}_i\left(P_{oi}[\mathcal{L}(\tilde{G})]\right) \subseteq L_{safe} = \overline{aba^*}$. The insertion function is thus i-enforcing.



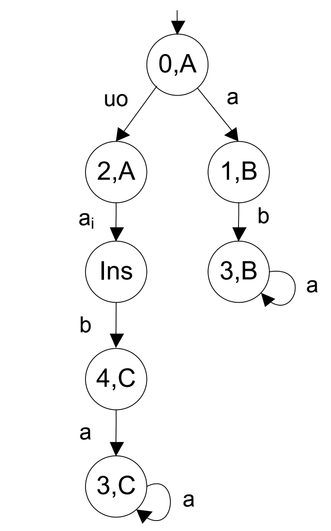
(a) The system G with $X_S = \{4\}$



(b) The estimator $\mathcal{E}$



(c) Mealy automaton M_i



(d) The equivalent automaton $\tilde{G}$

Figure 4.4: Automata used in Example IV.5

All manipulations in Algorithm 1 are polynomial in the number of states of the involved automata. $\tilde{G}$ has at most $|X||X_{ia}| + k|E_o||X||X_{ia}|$ states, where k is the largest number of inserted events per insertion action. The first term comes from the input parallel composition of G and IA. The second term exists because f_I inserts at most k events for every observable event at all states in G_i, and each of them requires an additional dummy state for unfolding G_i to $\tilde{G}$. Also, while $\tilde{G}$ is in general nondeterministic under the effect of map $\mathcal{M}_i \circ P_{oi}$, checking language inclusion of $\mathcal{M}_i\left(P_{oi}[\mathcal{L}(\tilde{G})]\right) \subseteq L_{safe}$ can be performed in polynomial time without determinizng $\tilde{G}$ because H is assumed to be deterministic.

4.5 Verifying I-enforceability of An Opacity Notion

Using the results in Section 4.4, we can verify if a given insertion function is i-enforcing with respect to the considered opacity notion. This problem inspires two further questions: (Q1) How to verify if a given opacity notion is i-enforceable, i.e., if there exists an i-enforcing insertion function? (Q2) How to synthesize an i-enforcing insertion function if one exists? We will answer both questions using a special automaton called the *All Insertion Structure* (AIS) constructed in this section. (Q1) will be answered in Section 4.5.2, and (Q2) will be answered in Section 4.6.

4.5.1 The Construction of the All-Insertion Structure (AIS)

The All Insertion Structure (AIS) is an automaton that enumerates, in a compact transition structure, *all deterministic i-enforcing insertion functions for a given secret of the system.* The whole construction comprises four stages: (1) the i-verifier; (2) the meta-observer; (3) the unfolded i-verifier; and (4) the AIS. There is a language equality check after stage (1). The remaining three stages are needed only when the language equality holds. In each of the four subsequent subsections, the structure of the corresponding stage will be defined and the algorithms will be provided.

Stage 1: The i-verifier V

The purpose of the *i-verifier* V is to identify all insertions that map any given string

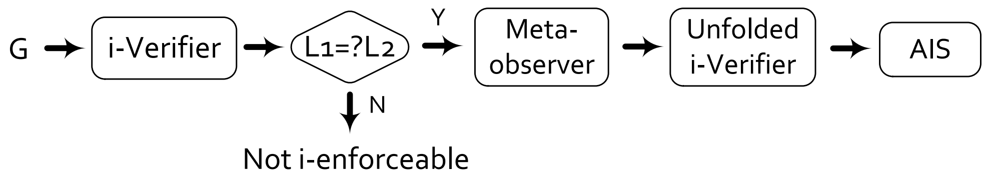


Figure 4.5: The construction flow of the AIS

to a safe string. For this purpose, we build the desired estimator $\mathcal{E}^d$, which generates the safe language L_{safe}, and the feasible estimator $\mathcal{E}^f$, which includes all possible insertions, and synchronize them with the *dashed parallel composition*. The resulting automaton represents an insertion function that is *nondeterministic*, *safe*, and *maximally-inserting*.

The desired estimator $\mathcal{E}^d$ and the feasible estimator $\mathcal{E}^f$ are obtained from the system estimator $\mathcal{E}$. To build $\mathcal{E}^d$ from $\mathcal{E}$, we delete in $\mathcal{E}$ all estimates that reveal the secret and take the accessible part. According to Proposition IV.1, $\mathcal{E}^d$ generates L_{safe}. To build $\mathcal{E}^f$, we add self-loops for every inserted event $e_i \in E_i$ (represented by dashed lines) at all states in $\mathcal{E}$. This estimator represents a nondeterministic insertion function that inserts the set of all possible insertions E_i^* upon every (s, e_o). Then, by dashed parallel composing the two estimators, the resulting automaton, called the i-verifier V, includes *all* possible insertions that map original strings to safe strings. Notice that a path in V is a sequence of alternating inserted strings and observable events. Such an alternating path, called an *insertion path*, shows how an insertion function inserts events as it receives observable events online from the system. Thus, having all insertion paths, the i-verifier is a representation of a nondeterministic insertion function. This nondeterministic insertion function, denoted by $\tilde{f}_I$, is safe and maximally-inserting because it inserts *all* strings that render a given original string safe.

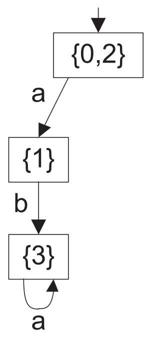
Formally, the i-verifier is an automaton denoted by $V = (M_v, E_o \cup E_i, \delta_v, m_{v,0})$. A state $m_v \in M_v$ is a pair of estimates (m_d, m_f), where m_d is a safe estimate from $\mathcal{E}^d$ and

m_f is the genuine (potentially) unsafe estimate from $\mathcal{E}^f$. In our opacity enforcement mechanism, m_d is the estimate of the intruder who eavesdrops with $\mathcal{M}_i$ and m_f is the estimate of the system designer who can distinguish inserted events with P_{und}. Although the genuine observed behavior of G leads to a possibly unsafe estimate m_f, the intruder is tricked into generating a safe estimate m_d from its observation.

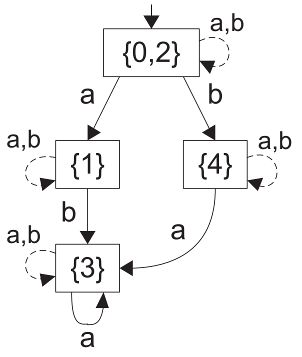
Example IV.6. Consider again G in Figure 4.4(a) where state 4 is the secret state. We want to enforce CSO using an insertion function and thus build the AIS to determine if CSO is i-enforceable. This example is our running example to illustrate the four stages of the construction of the AIS. Here we construct the i-verifier V.

We first build $\mathcal{E}^d$ in Figure 4.6(a) by removing m_3 from $\mathcal{E}$ and taking the accessible part. Then, we build $\mathcal{E}^f$ by adding self-loops for a_i and b_i at every state in $\mathcal{E}$, as shown in Figure 4.6(b); the inserted events a_i and b_i are represented by dashed lines. Afterwards, by dashed parallel composing $\mathcal{E}^d$ and $\mathcal{E}^f$, we obtain V shown in Figure 4.6(c). We use this example to see the features of the nondeterministic insertion function $\tilde{f}_I$. First, the safety property of $\tilde{f}_I$ can be shown by the transitions from (m_0, m_0) to (m_1, m_0) and then to (m_2, m_3), which correspond to inserting a_i before the first b. With this insertion, the intruder is tricked into thinking that G outputs ab and thus generates safe estimate m_2, while the original observed string b corresponds to the unsafe estimate m_3. Second, the nondeterminism of $\tilde{f}_I$ can be seen by the set $a_i b_i a_i^*$ inserted before the first a. Then, because all and only inserted strings in $(a_i b_i a_i)^*$ modify a to be safe strings, the insertion function is maximally-inserting.

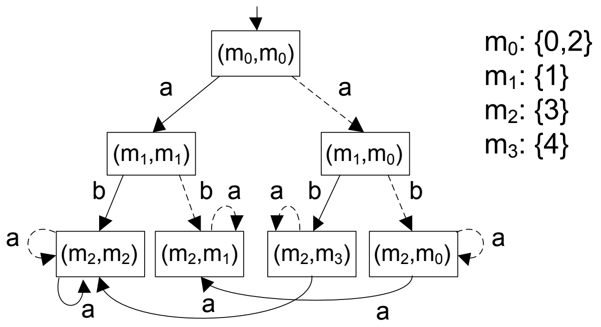
All strings output by $\tilde{f}_I$ are safe. However, $\tilde{f}_I$ may not respond to the full system output behavior $P[\mathcal{L}(G)]$; i.e., it may not be admissible. If $\tilde{f}_I$ is not admissible, then there is no deterministic insertion function that is safe and admissible. Therefore, opacity is not i-enforceable and the AIS does not exist. In other words, the admissibility of $\tilde{f}_I$ is a necessary condition for i-enforceability. This result is proven in



(a) The desired estimator $\mathcal{E}^d$



(b) The feasible estimator $\mathcal{E}^f$



(c) The i-verifier V

Figure 4.6: The system, the estimators, and the i-verifier used in Example IV.6.

Theorem IV.7.[2]

Theorem IV.7. *The considered opacity notion is not i-enforceable if*

$$P_{und}[\mathcal{L}(V)] \neq P[\mathcal{L}(G)] \tag{4.2}$$

Proof. First, V is a representation of $\tilde{f}_I$ that contains all possible safe insertions. Thus, all i-enforcing deterministic insertion functions must be included in V. Second, in V, dashed transitions are insertions. By applying projection P_{und}, we obtain the system behaviors that $\tilde{f}_I$ responds to, i.e., $P_{und}[\mathcal{L}(V)]$. If $P_{und}[\mathcal{L}(V)] \neq P[\mathcal{L}(G)]$, we know that $\tilde{f}_I$ does not respond to all observable behaviors of G. Thus, it is not admissible and not i-enforcing. If $\tilde{f}_I$ is not i-enforcing, no deterministic i-enforcing insertion function exists. The considered opacity notion is not i-enforceable. □

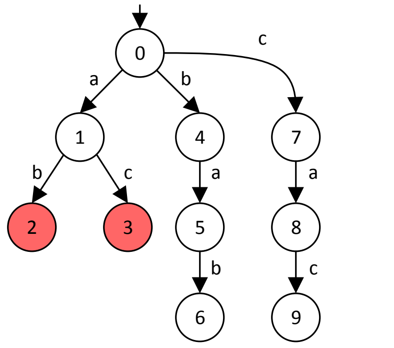
[2]This theorem was inadvertently omitted in [58].

If $P_{und}[\mathcal{L}(V)] \neq P[\mathcal{L}(G)]$, we know from Theorem IV.7 that opacity is not i-enforceable and we can stop the construction of the AIS. However, if $P_{und}[\mathcal{L}(V)] = P[\mathcal{L}(G)]$, we do not know whether opacity is i-enforceable or not. The existence of the nondeterministic i-enforcing insertion function $\tilde{f}_I$ does not imply the existence of a deterministic i-enforcing insertion function. Below we provide an example where no deterministic insertion function exists even if Equation (4.2) holds.

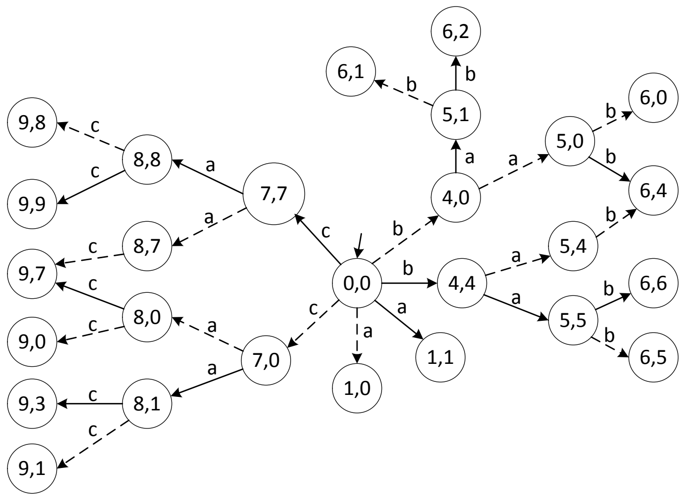
Example IV.8. Consider CSO of a system with $E_o = \{a, b, c\}$. We show directly the estimator in Figure 4.7(a), whose states are the estimates of system states; for simplicity, these states are numbered from 0 to 9. Assume that states 2 and 3 are unsafe and that the other states are safe. We want to enforce opacity using insertion functions and thus build the AIS to verify if CSO is i-enforceable. The i-verfier V is built in 4.7(b). One can verify that $P_{und}[\mathcal{L}(V)] = P[\mathcal{L}(G)]$ holds and thus the nondeterministic insertion function $\tilde{f}_I$ is i-enforcing. In Figure 4.7(b), we trace strings whose first solid transitions are labeled with "a" and see that $\tilde{f}_I$ insert two events $\{b, c\}$ before a, i.e., $\tilde{f}_I(\varepsilon, a) = \{b, c\}$. Inserting set $\{b, c\}$ makes ab look like non-secret bab (for state 2) and ac look like non-secret cac (for state 3). However, if we want to build a deterministic insertion function, the insertion function must choose to insert either $f_I(\varepsilon, a) = b$ or $f_I(\varepsilon, a) = c$. If we choose $f_I(\varepsilon, a) = b$, which corresponds to the path $(0, 0) \dashrightarrow (4, 0) \rightarrow (5, 1)$ in V, then the insertion function cannot respond to system string ac and thus it is not admissible. Similarly for the other case. Therefore, no deterministic i-enforcing insertion function exists. Opacity is not i-enforceable.

Stage 2: The meta-observer $mObs(V)$

Recall that V represents nondeterministic insertion function $\tilde{f}_I$. In V, insertions to the same system output may lie on different insertion paths. Therefore, grouping together all insertions responding to a given system output is needed before we enumerate all deterministic insertion functions. In this stage, we first focus on all insertions that



(a) The current-state estimator



(b) The i-verifier V

Figure 4.7: The system and the i-verifier in Example IV.8.

respond to past observed string s without considering the system's current output, i.e., $\cup_{e_o \in E_o} \tilde{f}_I(s, e_o)$. For this purpose, we build the so-called *meta-observer* of the i-verifier, $mObs(V)$. Let $Obs(V) = (M_{v,obs}, E_o, \delta_{v,obs}, m_{v,obs,0})$ be the standard observer automaton of V with respect to projection P_{und}. Given an i-verifier V, the meta-observer $mObs(V) = (M_{v,mo}, E_o \cup E_i, \delta_{v,obs}, \delta_{v,mo}, m_{v,mo,0})$ is a special observer of V, with respect to P_{und}, where the unobservable transitions (E_i) are preserved. A meta-observer state is defined to be a pair consisting of a state in V and a state in $Obs(V)$, $m_{v,mo} = (m_v, m_{v,obs}) \in M_{v,mo} = M_v \times M_{v,obs}$. In $mObs(V)$, there are two transition functions: $\delta_{v,mo}$ and $\delta_{v,obs}$. Transition function $\delta_{v,mo} : M_{v,mo} \times (E_o \cup E_i) \rightarrow M_{v,mo}$ captures the effect of insertion responses. Transition function $\delta_{v,obs} : M_{v,obs} \times E_o \rightarrow M_{v,obs}$, on the other hand, is inherited from $Obs(V)$ to capture the system's observed behavior. With both functions, we can keep the information of every insertion response while grouping them together. In Figure 4.8, we show the meta-observer for Example IV.6. Note that only $\delta_{v,mo}$, which is represented by the dashed and the solid transitions, is shown. The function $\delta_{v,obs}$ is not explicitly shown as it can be inferred from $\delta_{v,mo}$. The formal construction of $mObs(V)$ is presented in

Algorithm 2.

```
Algorithm 2: Construct the meta-observer of $V$
  input  : $V = (M_v, E_o \cup E_i, \delta_v, m_{v,0})$ and $Obs(V) = (M_{v,obs}, E_o, \delta_{v,obs}, m_{v,obs,0})$
           w.r.t $P_{und}$
  output: $mObs(V) = (M_{v,mo}, E_o \cup E_i, \delta_{v,obs}, \delta_{v,mo}, m_{v,mo,0})$
1 $m_{v,mo,0} = (m_{v,0}, m_{v,obs,0})$. Set $M_{v,mo} = \{m_{v,mo,0}\}$
2 for $\beta = (b, b_{obs}) \in M_{v,mo}$ whose dashed descendants have not been expanded do
      for $e \in E_i$ do
          if $\delta_v(b, e)$ is defined then
              $\delta_{v,mo}(\beta, e) = (\delta_v(b, e), b_{obs})$
              Add $\delta_{v,mo}(\beta, e)$ to $M_{v,mo}$
3 for $\beta = (b, b_{obs}) \in M_{v,mo}$ whose dashed descendants have been expanded but
  whose solid descendants have not been expanded do
      for $e \in E_o$ do
          if $\delta_v(b, e)$ is defined then
              $\delta_{v,mo}(\beta, e) = (\delta_v(b, e), \delta_{v,obs}(b_{obs}, e))$
              Add $\delta_{v,mo}(\beta, e)$ to $M_{v,mo}$
4 Go back to step 2 and repeat until all nodes in $M_{v,mo}$ have been completely
  expanded
5 Add $\delta_{v,obs}$ from $Obs(V)$ to $mObs(V)$
```

Example IV.9. Given V in Figure 4.6(c), we follow Algorithm 2 and build the meta-observer $mObs(V)$ in Figure 4.8. In step 1, the initial state is $((m_0, m_0), A)$, where (m_0, m_0) is the initial-state of V and $A = \{(m_0, m_0), (m_1, m_0), (m_2, m_0)\}$ is the initial state of $Obs(V)$. In step 2, $((m_0, m_0), A)$ expands to $((m_1, m_0), A)$ and then $((m_1, m_0), A)$ expands to $((m_2, m_0), A)$. This dashed-descendant expansion generates all $M_{v,mo}$ states whose $M_{v,obs}$ part is also A; these $M_{v,mo}$ states belong to the topmost group in the figure. In step 3, $((m_0, m_0), A)$ expands to $((m_1, m_1), B)$ with event a, $((m_1, m_0), A)$ expands to $((m_2, m_3), C)$ with b, and $((m_2, m_0), A)$ expands to $((m_2, m_1), B)$ with a. This time, the solid-descendant expansion generates all $M_{v,mo}$ states reached with E_o from the existing states. Each newly generated $M_{v,mo}$ state has a $M_{v,obs}$ part that is different from A. We iterate between step 2 and step 3 until all $M_{v,mo}$ state have been completely expanded. Finally, we complete constructing

$mObs(V)$ by adding transition function $\delta_{v,obs}$ from $Obs(V)$.

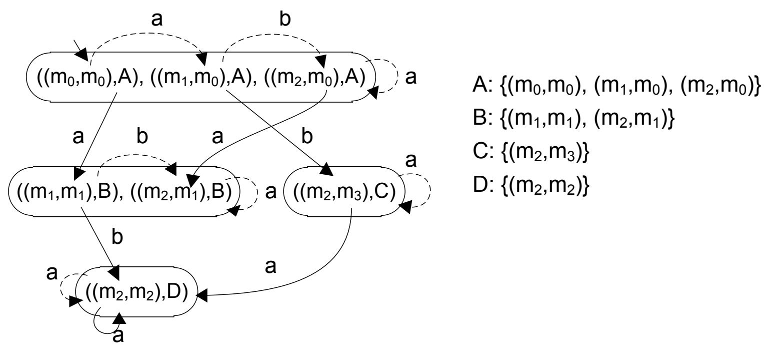


Figure 4.8: The meta-observer of the i-verifier, $mObs(V)$. The solid ellipses around states show the states of $Obs(V)$.

Stage 3: The unfolded i-verifier V_u

In Stage 2, the meta-observer groups together all insertion strings for every given s; i.e., $\cup_{e_o \in E_o} \tilde{f}_I(s, e_o)$. In this stage, we want to list all insertion strings in $\tilde{f}_I(s, e_o)$ for any given (s, e_o) and then enumerate all deterministic insertion functions. We "unfold" all deterministic insertion functions from $mObs(V)$, and build the so-called *unfolded i-verifier* V_u.

The unfolded i-verifier V_u is a game structure played between the "system player" G and the "insertion-function player" I. The initial state of V_u is played by G. The two players alternate turns in the game. All outgoing transitions of a given state are *actions* of the player who moves at the state. Hence, all deterministic safe insertion functions are enumerated with sequences of alternating *actions*. More specifically, actions of G are output events from the system and actions of I are safe insertion strings. A sequence of actions is an insertion path that tell us the following: Given that player G has output s and is outputting e_o, player I can insert s_I before e_o.

Formally, V_u is a bipartite graph that is defined as an automaton $V_u = (Y \cup Z, E_o \cup M_{v,mo}, f_{yz} \cup f_{zy}, y_0)$. An example of V_u is shown in Figure 4.9. We interpret V_u as a two-player game structure. The first player is G that moves at Y (square-

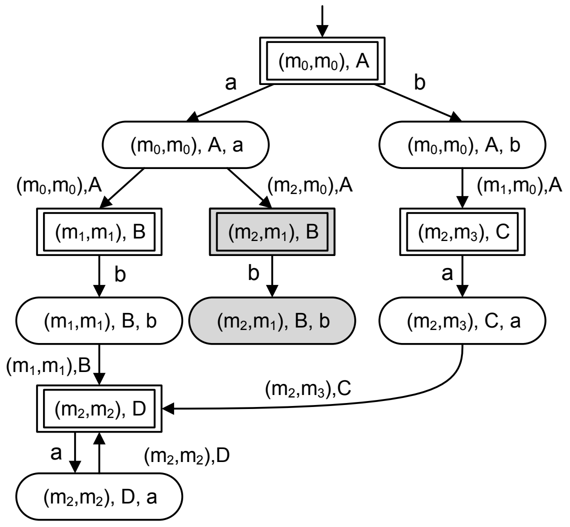


Figure 4.9: The unfolded i-verifier V_u built from $mObs(V)$ in Figure 4.8. The shaded states are pruned when constructing the AIS.

shaped) states; the second player is player I that moves at Z (ellipse-shaped) states. Y and Z states are information states of players G and player I, respectively. That is, each state contains enough information for the corresponding player to enumerate its actions. More specifically, a state $y \in Y$ is a meta-observer state, $Y = M_{v,mo} \subseteq M_v \times M_{v,obs}$ and each action at y is an output event $e \in E_o$ from the system. On the other hand, a state $z \in Z$ consists of its predecessor Y state and the action of observable event e that player G has just made, $Z \subseteq Y \times E_o = M_{v,mo} \times E_o$. Each action at $z = (m, e) \in Z$ is a state $m' \in M_{v,mo}$ in $mObs(V)$ that compactly represents a set of insertion strings given by function $Ins(m, m') = \{s_I \in E_i^* : \delta_{v,mo}(m, s_I) = m'\}$ where $m, m' \in M_{v,mo}$. Take the V_u in Figure 4.9 for example. At $z = (((m_0, m_0), A), a)$, action $((m_2, m_0), A)$ represents a set of insertion strings given by $Ins\,(((m_0, m_0), A), ((m_2, m_0), A)) = \{s_I \in E_i^* : \delta_{v,mo}(((m_0, m_0), A), s_I) = ((m_2, m_0), A)\} = a_i b_i a_i^*$. The transition function from Y to Z is denoted by $f_{yz} : Y \times E_o \to Z$, and the transition function from Z to Y is denoted $f_{zy} : Z \times M_{v,mo} \to Y$. As the system is the first player, the initial state of V_u is defined as $y_0 = m_{v,mo,0}$. The formal construction of V_u is presented in Algorithm 3.

```
Algorithm 3: Construct $V_u$
  input  : $mObs(V) = (M_{v,mo}, E_o \cup E_i, \delta_{v,obs}, \delta_{v,mo}, m_{v,mo,0})$
  output: $V_u = (Y \cup Z, E_o \cup M_{v,mo}, f_{yz} \cup f_{zy}, y_0)$
1 $y_0 = m_{v,mo,0}$. Set $Y = \{y_0\}$
2 for $y = (m_v, m_{v,obs}) \in Y$ that have not been examined do
      for $e \in E_o$ do
          if $\delta_{v,obs}(m_{v,obs}, e)$ is defined then
              $f_{yz}(y, e) = (y, e)$
              Add $f_{yz}(y, e)$ to $Z$
3 for $z = (m_{v,mo}, e) \in Z$ that have not been examined do
      for $m \in M_{v,mo}$ do
          if $\exists s_I \in E_i^*$ such that $m = \delta_{v,mo}(m_{v,mo}, s_I)$ and $\delta_{v,mo}(m, e)$ is defined
          then
              $f_{zy}(z, m) = \delta_{v,mo}(m, e)$
              Add $f_{zy}(z, m)$ to $Y$
4 Go back to step 2 and repeat until all accessible part has been built
```

Stage 4: The All Insertion Structure (AIS)

The i-verifier enumerates all deterministic insertion functions in $\tilde{f}_I$. These insertion functions, however, may not be all admissible. In Stage 4, we prune away inadmissible insertion paths in V_u and build the All Insertion Structure (AIS). An insertion path is not admissible if it cannot respond to every system output.

Recall that, in V_u, player G plays at Y states and player I plays at Z states. If there is a state $z \in Z$ that has no outgoing action (i.e., z is a deadlocked state), then the insertion-function player cannot respond to the system player when it reaches z. That is, z belongs to an inadmissible insertion path and thus should be pruned away. When we prune away z, we must also prune away its incoming actions. However, its incoming actions should not be pruned because they are actions of player G. Thus, we have to prune some earlier insertion actions in order to prevent the AIS from generating this deadlocked state z. But such earlier pruning may create other deadlocked Z states. Hence, this process requires an iterative pruning algorithm until no deadlocked states exist. We use the following example to illustrate how we prune V_u to obtain the AIS.

Example IV.10. Consider again the V_u in Figure 4.9. State $(((m_2, m_1), B), b)$ is a deadlocked Z state. All insertion paths leading to $(((m_2, m_1), B), b)$ are inadmissible because no insertion is available when the system outputs event b. To prune such inadmissible insertion paths, we prune $(((m_2, m_1), B), b)$ and its incoming action b. However, we cannot remove b because it is the action of the system at state $((m_2, m_1), B)$ and insertion functions have no control of system actions. Therefore, to completely prune away the deadlocked state $((m_2, m_1), B)$, an insertion function should decide not to respond with $((m_2, m_0), A)$ earlier at $(((m_0, m_0), A), a)$. The resulting AIS is the structure in Figure 4.9 without the shaded states.

In the above pruning process, we want to prune away inadmissible insertion paths and only prune away such paths when necessary. Formally, we formulate this process as an instance of the "Basic Supervisory Control Problem - Nonblocking Case" (BSCP-NB), in the terminology of [10]. For this purpose, marked states and controllable/uncontrollable events need to be defined in V_u. We mark all Y states in V_u and leave all Z states unmarked. This is because an insertion function completes insertions at Y states, but is choosing an insertion response at Z states. We model all outgoing actions of Y states (E_o) as uncontrollable and those of Z states ($M_{v,mo}$) as controllable because an insertion function has no control of the system output but is able to choose an insertion response. The specification for the supervisory control problem is the trimmed automaton of V_u, defined by V_u^{trim}. Finally, the AIS is the minimally restrictive nonblocking supervisor of V_u (see [39, 10]) for this BSCP formulation. By construction, the AIS is a sub-automaton of V_u. This is because the specification automaton in this instance of BSCP-NB is V_u^{trim}. The formal construction is presented in Algorithm 4.

Note that it is possible that Algorithm 4 returns the empty automaton when

```
Algorithm 4: Construct the AIS
    input  : $V_u = (Y \cup Z, E_o \cup M_{v,mo}, f_{yz} \cup f_{zy}, y_0)$
    output: AIS= $(Y \cup Z, E_o \cup M_{v,mo}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$
  1 Mark all the $Y$ states in $V_u$
  2 Let $E_o$ be uncontrollable and $M_{v,mo}$ be controllable
  3 Trim $V_u$ and let $V_u^{trim}$ be the specification automaton
  4 Obtain $[\mathcal{L}_m(V_u^{trim})]^{\uparrow C}$ w.r.t $\mathcal{L}(V_u)$ by following the standard $\uparrow C$ algorithm in
    [10]
  5 Return the resulting automaton representation of $[\mathcal{L}_m(V_u^{trim})]^{\uparrow C}$, which is a
    sub-automaton of $V_u^{trim}$
```

$[\mathcal{L}_m(V_u^{trim})]^{\uparrow C} = \emptyset$. Because the pruning procedure prunes away all inadmissible insertion paths, the resulting AIS enumerates all i-enforcing insertion functions. That is, if we use the AIS to track the system output and choose a random insertion at every Z state, we can synthesize any i-enforcing insertion function.[3] We will prove this in Theorem IV.13 using the following two lemmas. In these proofs, we always have $P_{und}[\mathcal{L}(V)] = P[\mathcal{L}(G)]$ because this is a necessary condition for the existence of the AIS.

Lemma IV.11. *An insertion function can be synthesized from the AIS if it is i-enforcing.*

Proof. Given an i-enforcing insertion function f_I, it is included in V because $\tilde{f}_I$ is the maximally-inserting nondeterministic i-enforcing insertion function. When building $mObs(V)$, we simply group transitions of V and thus no transitions are lost or added. That is, no insertion paths are lost or added. When we build V_u, the only dashed strings that will be excluded are those not responding to an output. Also, when building the AIS, we only remove states that are inevitably leading to a state where no insertions are available. None of the above automata remove an admissible path from its previous automaton. Therefore, all insertion paths of f_I, included in V at the beginning, must still exist in the AIS. Thus, f_I can be synthesized from the AIS. $\square$

[3]We need not choose the same insertion if we reach the same z from different paths. Thus, the synthesized insertion function can have an infinite string-based domain while it is synthesized from a finite structure.

Lemma IV.12. *If an insertion function is synthesized from the AIS, it must be i-enforcing.*

Proof. Given an insertion function f_I synthesized from the AIS, we want to prove that it is i-enforcing by contradiction. Let us assume that this insertion function is not i-enforcing. That is, f_I is not safe or not admissible. But because f_I is synthesized from the AIS, it must be safe because the AIS is constructed from V and V only includes safe insertions. Therefore, f_I can only be inadmissible. If f_I is not admissible, it must have an insertion path that leads to a deadlocked Z state where no insertions are available. But by construction, the AIS cannot have such a path. This is a contradiction. Therefore, f_I should be i-enforcing. □

Theorem IV.13. *The AIS enumerates all and only i-enforcing insertion functions.*

Proof. Follows from Lemmas IV.11 and IV.12. □

4.5.2 Verification of I-enforceability Property

Let us now go back to question (Q1) posed at the beginning of this section: How to verify if a given opacity notion is i-enforceable? As the AIS enumerates all and only i-enforcing insertion functions, it can be used to determine the i-enforceability property of a given opacity notion. In fact, we can verify i-enforceability by checking if the AIS is the empty automaton or not.

Theorem IV.14. *Opacity is i-enforceable if and only if the AIS is not the empty automaton.*

Proof. (Only if) If opacity is i-enforceable, then we can find an i-enforcing insertion function. From Lemma IV.11, we know that this insertion function can be synthesized from the AIS. Therefore, the AIS cannot be the empty automaton. (If) If the AIS is not the empty automaton, we can synthesize an i-enforcing insertion function from the AIS based on Lemma IV.11. Therefore, the opacity notion must be i-enforceable. □

Now, let us go back to the system in Figure 4.7(a) and see how we obtain the empty AIS when opacity is not i-enforceable.

Example IV.15. In Example IV.8, we concluded that opacity is not i-enforceable by reasoning on the insertions in V. In this example, we will show that the AIS is the empty automaton. Shown in Figure 4.10 is the unfolded i-verifier V_u of the system. We trim in V_u the deadlocked states and get the specification automaton V_u^{trim}, which is shown in Figure 4.10 without the shaded states. To build the AIS, we obtain $[\mathcal{L}_m(V_u^{trim})]^{\uparrow C}$ w.r.t $\mathcal{L}(V_u)$. States $((5,1),B),((8,1),B),((1,1),B)$ will be deleted in the $\uparrow C$ algorithm because each of them leads to an illegal state with an uncontrollable event. The deletion of these three states makes the insertion state $(((0,0),A),a)$ deadlocked. Thus, in the next iteration of the $\uparrow C$ algorithm, state $(((0,0),A),a)$ needs to be pruned. However, pruning $(((0,0),A),a)$ violates the controllability condition again. As a result, we need to prune away the initial state and thus obtain the empty automaton.

4.6 Synthesis of I-enforcing Insertion Functions

We now go back to question (Q2) posed in Section 4.5 and use the AIS to synthesize an i-enforcing insertion function. Such an insertion function, encoded as an insertion automaton (IA), is synthesized by selecting transitions and states in the AIS. The synthesis algorithm is formally shown in Algorithm 5.

In step 2.b of Algorithm 5, the question of which m' and s_I to choose arises. We can establish a selection criterion by designing an appropriate cost function, and formulate an optimal control problem. Because the AIS considers *all* i-enforcing insertion functions, it provides a structure over which such an optimal control problem can be formulated and solved. This is the objective of the next chapter. In this chapter, we only synthesize *one* i-enforcing insertion function.

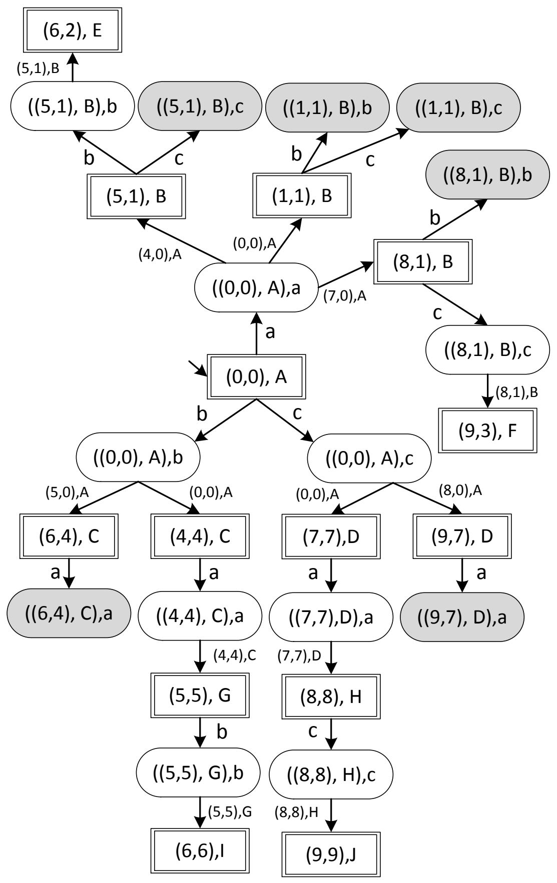


Figure 4.10: V_u of Example IV.8. The shaded states are pruned in V_u^{trim}.

One can run Algorithm 5 as long as the AIS is not the empty automaton. By following Algorithm 5, we will always obtain a finite-state insertion automaton (i.e., a finite i-enforcing insertion function). As a result, it is sufficient and without loss of generality to consider only finite insertion functions in the synthesis.

Corollary IV.16. *There exists a finite i-enforcing insertion function if the considered opacity notion is i-enforceable.*

Proof. If opacity is i-enforceable, then the AIS is not the empty automaton, according to Theorem IV.14. Hence, a finite i-enforcing insertion function can be synthesized by following Algorithm 5. □

Example IV.17. Given the AIS in Figure 4.9, we construct an insertion automaton

Algorithm 5: Synthesize an insertion automaton

```
    input  : AIS $= (Y \cup Z, E_o \cup M_{v,mo}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$
    output: $IA = (X_{ia}, E_o, E_i^* E_o, f_{ia}, q_{ia}, x_{ia,0})$
1   Let $x_{ia,0} = y_0$. Set $X_{ia} = \{x_{ia,0}\}$
2   for $x_{ia} \in X_{ia}$ that have not been examined do
2.a     if $x_{ia} \in Y$ then
            for $e \in E_o$ do
                if $f_{AIS,yz}(x_{ia}, e)$ is defined then
                    $f_{ia}(x_{ia}, e) = f_{AIS,yz}(x_{ia}, e)$ i
                    Add $f_{ia}(x_{ia}, e)$ to $X_{ia}$
2.b     else if $x_{ia} = (m, e) \in Z$ then
            Select one $m' \in M_{v,mo}$ for which $f_{AIS,zy}(x_{ia}, m')$ is defined
            Select one string $s_I \in Ins(m, m')$
            $f_{ia}(x_{ia}, s_I) = f_{AIS,zy}(x_{ia}, m')$
            Add $f_{ia}(x_{ia}, s_I)$ to $X_{ia}$
3   for $x_{y1}, x_z, x_{y2} \in X_{ia}$ where $x_{y1}, x_{y2} \in Y, x_z \in Z$ and $f_{ia}(x_{y1}, e) = x_z$,
    $f_{ia}(x_z, s_I) = x_{y2}$ do
        Remove $x_z$ from $X_{ia}$
        Redefine $f_{ia}$ and define $q$ such that $f_{ia}(x_{y1}, e) = x_{y2}$ and $q_{ia}(x_{y1}, e) = s_I e$
```

by following Algorithm 5. In the AIS, every Z state has only one outgoing action. Thus, in step 2.b we only need to choose one string for every $M_{v,mo}$ action. We choose to greedily insert the least number of events for every $M_{v,mo}$ action. For action $((m_1, m_0), A)$, we choose a_i. For the other actions, we choose ε. In step 3, we remove all Z states and combine actions. Therefore, $((m_0, m_0), A) \xrightarrow{b} (((m_0, m_0), A), b) \xrightarrow{a_i} ((m_2, m_3), C)$ becomes $((m_0, m_0), A) \xrightarrow{b/a_i b} ((m_2, m_3), C)$. The resulting IA is shown in Figure 4.11.

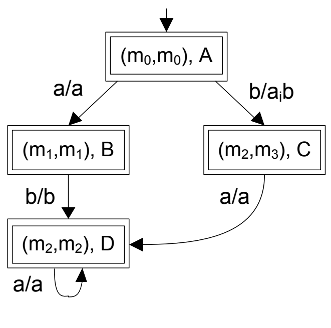


Figure 4.11: An insertion automaton built from the AIS

4.7 Complexity of the Construction of the AIS

In this section, we discuss the computational complexity of the AIS. Recall that the AIS is constructed in four stages. To understand the complexity of each stage, we calculate the worst-case space complexity of each structure in terms of its previous structure. With no loss of generality, we consider current-state opacity. Given a system G with $|X|$ states, verifying CSO requires building the current-state estimator (i.e., the observer automaton of G), which has at most $|X_{obs}| = 2^{|X|}$ states. To build the AIS, in stage 1, we build the i-verifier V. The state space of the i-verifier is $|X_v| = |X_{obs}|^2$ because V is the dashed parallel composition of two special current-state estimators. In stage 2, the number of states in the meta-observer $mObs(V)$ is at most $|X_{mobs}| = |X_v|2^{|X_v|}$ since $M_{v,mo} = M_v \times M_{v,obs}$. Then, in stage 3, we "unfold" $mObs(V)$ to obtain the unfolded i-verifier V_u. The unfolded i-verifier has $|X_{v_u}| = |X_{mobs}| + |E_o||X_{mobs}|$ states in the worst case, where the first and the second terms account for information states of the system and the insertion function, respectively. Lastly, the AIS has at most $|X_{v_u}|$ states because the AIS is the recognizer for $\mathcal{L}_m(V_u^{trim})^{\uparrow C}$. Overall, the space complexity of all the structures built in obtaining the AIS is $O(2^{|X_{obs}|^2})$.

In the above discussion, each worst-case explosion is the theoretical upper bound. In practice, the average complexity may be much smaller. To gain better insight, we conducted an empirical study of the construction of the AIS. A program that generates random automata was implemented. A random automaton is constructed upon the input of the number of states $|X|$, the number of observable events $|E_o|$, and the number of secret states $|X_S|$. The program first generates an $|X|$-state connected directed graph by iteratively adding a transition from an existing state to the newly-created state. Then, it generates more transitions so that the number of outgoing transitions for each state is uniformly distributed between 0 and $|E_o|$. We chose $\mu = 0.2$ to be the ratio of unobservable transitions in our experiments.

Among all transitions, μK transitions are randomly selected to be unobservable, where K is the total number of transitions. Next, given a state with k observable outgoing transitions, the program randomly selects k observable events to label these transitions. The resulting automaton is guaranteed to be deterministic. Finally, $|X_S|$ secret states are randomly selected.

Three experiments were conducted. The first experiment studies how i-enforceability is affected by the number of secret states. We randomly generated 100 10-state automata with $|E_o| = 3$ for $|X_S| = 1, 2, 5, 10$. Table 4.1 shows the number of automata for four outcomes: (i) opaque; (ii) i-enforceable; (iii) not i-enforceable with $P_{und}[\mathcal{L}(V)] \neq P[\mathcal{L}(G)]$; and (iv) not i-enforceable with the empty AIS. As expected, the number of opaque automata decreases as $|X_S|$ increases. When $|X_S| = 10$, i.e., all states are secret, all system behaviors are secret and thus the system cannot be i-enforceable. Also, when all states are secret, $\mathcal{E}^d$ and thus V are the empty automaton. Thus, opacity cannot be i-enforceable with $P_{und}[\mathcal{L}(V)] \neq P[\mathcal{L}(G)]$. Then, let us consider only the automata that are not opaque. The ratio of i-enforceable automata decreases as $|X_S|$ increases. We interpret this result as follows: the more likely the system reveals the secret, the more difficult it is to enforce the secret to be opaque.

Table 4.1: Experimental results of i-enforceability w.r.t $|X_S|$. Number of automata for four outcomes are recorded: (i) opaque; (ii) i-enforceable; (iii) not i-enforceable with $P_{und}[\mathcal{L}(V)] \neq P[\mathcal{L}(G)]$; and (iv) not i-enforceable with the empty AIS

$|X_S|$	(i)	(ii)	(iii)	(iv)
1	58	34	8	0
2	23	59	18	0
5	0	54	46	0
10	0	0	100	0

The second experiment studies how i-enforceability is affected by the number of events. Again, 100 10-state random automata were generated. We fixed the number of secret states $|X_S| = 1$ and varied the number of observable events $|E_o| = 1, 3, 5, 10$.

Table 4.2 shows the number of automata for the four aforementioned outcomes. The number of opaque automata increases as $|E_o|$ increases. As $|E_o|$ grows, the total number of events also grows. More events means that more different behaviors can be generated. With a wider range of system behaviors, it is more likely for a secret behavior to look like some non-secret behavior and thus more automata are opaque. Considering the fraction of i-enforceable automata among all non-opaque automata, we see that this fraction increases as $|E_o|$ increases. As there are more events that can be inserted, it is more likely for an insertion function to make a secret behavior look like a non-secret behavior.

Table 4.2: Experimental results of i-enforceability w.r.t $|E_o|$

$\|E_o\|$	(i)	(ii)	(iii)	(iv)
1	0	0	100	0
3	23	59	18	0
5	54	40	6	0
10	73	25	2	0

None of the above experiments resulted in an automaton that is not i-enforceable because the AIS is empty. This shows that if an automaton is not i-enforceable, it is likely that we can make an early conclusion by testing $P_{und}[\mathcal{L}(V)] \neq P[\mathcal{L}(G)]$ after stage 1. Therefore, the complexity of checking i-enforceability may be much smaller, depending on the structure of the automaton.

The third experiment studies the complexity in each stage of the construction of the AIS. We obtained 20 random automata with $|X| = 10, 20, 30$ by running the random automaton generation program and discarding opaque automata. In order to remove the complexity's dependencies between $|X_S|$ and $|E_o|$, we fixed the ratio of secret states such that $|X_S|/|X| = 0.1$ and scaled the number observable events such that $|E_o| = 3\left(\log \frac{|X|}{10} + 1\right)$. The scaling function for $|E_o|$ is motivated by the consideration that the state space grows exponentially and the event size grows linearly with the number of systems in composition. Tables 4.3 shows the average number of states in $Obs(G)$, V, $mObs(V)$, V_u, and the AIS.

Table 4.3: Experimental results of the complexity of the AIS

$\lvert X\rvert$	$\lvert X_{obs}\rvert$	$\lvert X_v\rvert$	$\lvert X_{mobs}\rvert$	$\lvert X_{v_u}\rvert$	$\lvert X_{AIS}\rvert$
10	25.3	723.1	729.7	1271.1	1262.3
20	115.8	14615.5	14820.2	22542.4	22443.6
30	357.4	142632.3	151165.3	212401.6	203473.4

Recall that the theoretical complexity of $|X_{AIS}|$ is $O(2^{|X_{obs}|^2})$. Table 4.3 shows that the average complexity in our experiments is far lower. This difference mainly comes from overestimating the complexity of meta-observers. The average state space cardinality of meta-observers in our experiments is only slightly greater than $|X_v|$, while it is $O(|X_v|2^{|X_v|})$ in theory. If we approximate $|X_{mobs}|$ to be $|X_v|$, then the complexity of $|X_{AIS}|$ becomes $O(|X_{obs}|^2)$, which is closer to our results in Table 4.3.

4.8 A More Compact AIS

So far, we have constructed an AIS, in a four-stage algorithm, that requires exponential complexity in the number of states of the state estimator used to verify opacity. In this section, we present a more compact AIS that embeds all i-enforcing insertion functions using fewer states. The proposed new AIS is constructed in a three-stage algorithm. Its state space is reduced to polynomial (in the number of states of the state estimator), thereby significant computational gains are achieved. More specifically, the three-stage construction procedure, as compared to the four-stage construction procedure in Section 4.5.1, omits the language check in Theorem IV.7 and the construction of the meta-observer $mObs(V)$; it also modifies the algorithm for constructing V_u. It comprises: (1) the i-verifier V; (2) the unfolded i-verifier V_u; and (3) the AIS. To distinguish the two construction algorithms, we will call the four-stage algorithm as the *exponential algorithm* and the three-stage algorithm as the *polynomial algorithm*, for the former requires exponential complexity and the latter requires polynomial complexity. Also, we denote the unfolded i-verifier and the AIS in the exponential algorithm by V_u^e and AIS_u^e, and those in the polynomial algorithm by V_u^p and AIS_u^p. The construction of the i-verifier V remains the same. We now formally present the polynomial algorithm

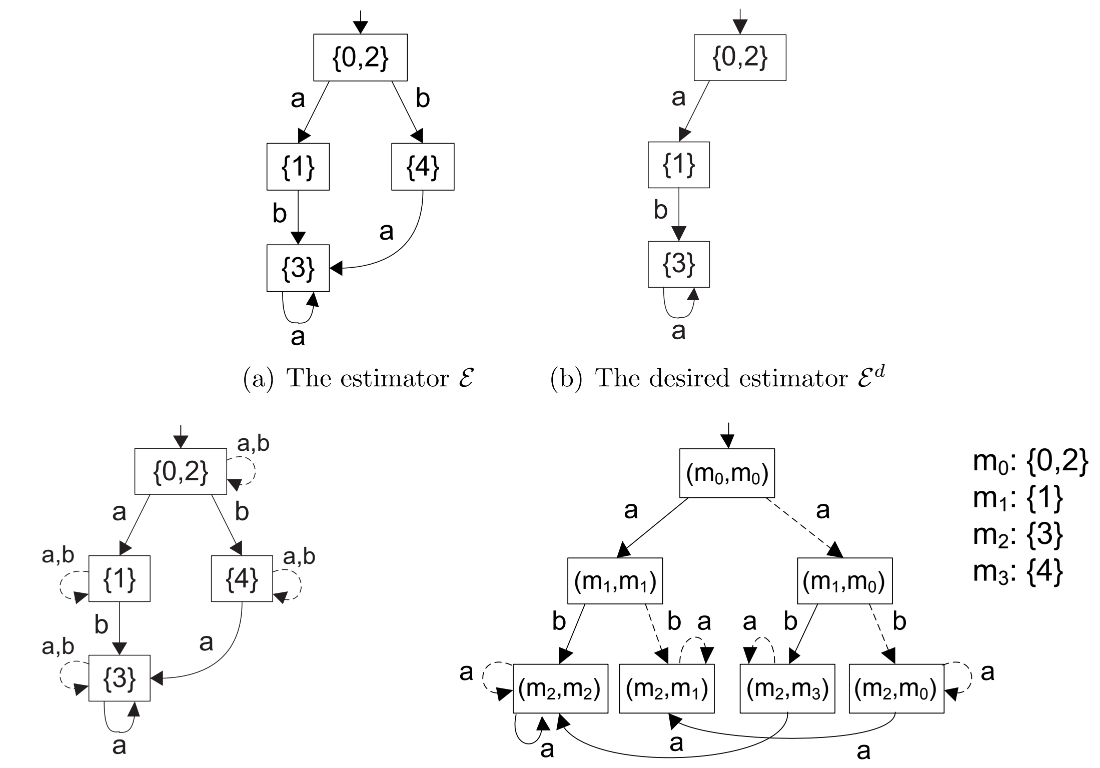
Stage 1: The i-verifier V

The construction of the i-verifier in the polynomial algorithm is the same as that instructed in the exponential algorithm. To make this section self-contained, we briefly recall the construction of the i-verifier. Consider the forward estimator that verifies opacity $\mathcal{E} = (X_\mathcal{E}, E_o, f_\mathcal{E}, x_{\mathcal{E},0})$. We build the desired estimator $\mathcal{E}^d$ by deleting in $\mathcal{E}$ all estimates that reveal the secret and taking the accessible part. Also, we build the feasible estimator $\mathcal{E}^f$ by adding self-loops for every inserted event $e_i \in E_i$ at all states in $\mathcal{E}$. Then, we construct the i-verifier by dashed parallel composing the two special state estimators; i.e., $V := \mathcal{E}^d||_d\mathcal{E}^f = (M_v, E_o \cup E_i, \delta_v, m_{v,0})$. In Figure 4.12, we show again $\mathcal{E}^d$, $\mathcal{E}^f$, and V in the running example in Section 4.5 (Example IV.6). We will reuse this example to present the polynomial algorithm.

Stage 2: The unfolded i-verifier V_u^p

Having V that identifies all safe insertions, we now build the "unfolded i-verifier" V_u^p directly from V. Note that we omit the language check in Theorem IV.7 and the construction of the meta-observer $mObs(V)$ in the exponential algorithm. In the polynomial algorithm, we rely on $\mathcal{E}$ and V to enumerate all deterministic safe insertion functions.

Just like V_u^e constructed in Section 4.5, the unfolded i-verifier V_u^p is a game structure played between the "system player" G and the "insertion-function player" I. Formally, V_u^p constructed in the polynomial algorithm is a bipartite graph that is defined as an automaton $V_u^p = (Y \cup Z, E_o \cup M_v, f_{yz} \cup f_{zy}, y_0)$. Shown in Figure 4.13



(a) The estimator $\mathcal{E}$ (b) The desired estimator $\mathcal{E}^d$

(c) The feasible estimator $\mathcal{E}^f$ (d) The i-verifier V

Figure 4.12: The system, the estimators, and the i-verifier V

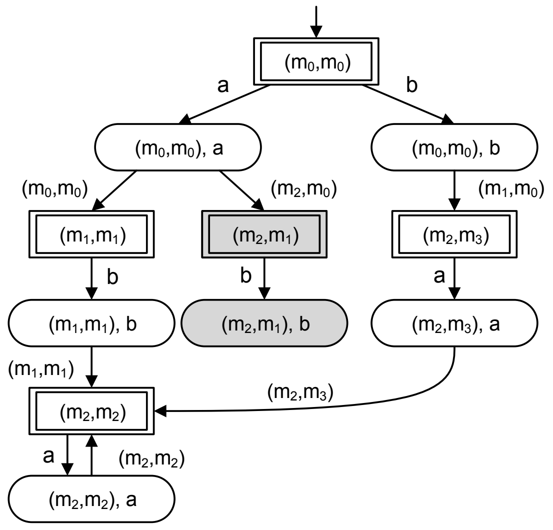


Figure 4.13: The unfolded i-verifier V_u^p. The shaded states are pruned when constructing the AIS^p.

is V_u^p built from the $\mathcal{E}$ and V in Figure 4.12. The first player is G that moves at Y (square-shaped) states; the second player is player I that moves at Z (ellipse-shaped) states. Y and Z states are information states of players G and I, respectively. That is, each state contains enough information for the corresponding player to enumerate its actions. The differences between V_u^p and V_u^e are the information contained in each Y and Z state. Here, in the polynomial algorithm, each $y \in Y$ state is a state $m = (m_d, m_f) \in M_v$ in V and each action at y is an output event $e \in E_o$ from the system. Because m_f is the real state estimate of the system, we can examine all event transitions from m_f in $\mathcal{E}$ to enumerate actions for player G. On the other hand, each Z state, denoted by $z = (y, e)$, consists of its predecessor state y and the action of observable event e that player G has just made. Because a Y state is also an M_v state in V, we also write $z = (m, e)$. Each action at $z = (m, e)$ is a state $m' \in M_v$ in V that compactly represents a set of inserted strings given by function $Ins(m, m') = \{s_I \in E_i^* : \delta_v(m, s_I) = m'\}$ where $m, m' \in M_v$. To enumerate all safe insertion strings, we can search on V using m and e. The transition function from Y to Z is denoted by $f_{yz} : Y \times E_o \to Z$, and the transition function from Z to Y

is denoted $f_{zy} : Z \times M_v \to Y$. As the system is the first player, the initial state of V_u^p is defined as $y_0 = m_{v,0}$. The formal procedure for constructing V_u^p is presented in Algorithm 6.

Algorithm 6: Construct V_u^p

```
  input  : $V = (M_v, E_o \cup E_i, \delta_v, m_{v,0})$ and $\mathcal{E} = (X_\mathcal{E}, E_o, f_\mathcal{E}, x_{\mathcal{E},0})$
  output: $V_u^p = (Y \cup Z, E_o \cup M_v, f_{yz} \cup f_{zy}, y_0)$
1 $y_0 = m_{v,0}$. Set $Y = \{y_0\}$
2 for $y = m_v = (m_d, m_f) \in Y$ that have not been examined do
      for $e \in E_o$ do
          if $f_\mathcal{E}(m_f, e)$ is defined then
              $f_{yz}(y, e) := (y, e)$
              Add $f_{yz}(y, e)$ to $Z$
3 for $z = (y, e) = (m, e) \in Z$ that have not been examined do
      for $m' \in M_v$ do
          if $\exists s_I \in E_i^*$ such that $m' = \delta_v(m, s_I)$ and $\delta_v(m', e)$ is defined then
              $f_{zy}(z, m') := \delta_v(m', e)$
              Add $f_{zy}(z, m')$ to $Y$
4 Go back to step 2 and repeat until all accessible part has been built
```

Step 2 can be done in $O(|M_v||E_o|)$ time. In step 3, we can compute and store the dashed connectivity for every $m, m' \in M_v$ in $O(|M_v|^3)$ time using Floyd-Warshall algorithm, and then determine all insertion actions in $O(|M_v|^2|E_o|)$ iterations. When $|E_o| < |M_v|$, this part can be done in $O(|M_v|^3)$ time. In all, V_u^p has at most $(|E_o| + 1)|M_v|$ states.

Stage 3: The All Insertion Structure (AIS^p)

Once we obtain V_u^p, we can prune the inadmissible insertions and obtain the AIS^p in the same way instructed in the exponential algorithm. The formal algorithm is presented in Algorithm 7. This is the same as Algorithm 4 in the exponential algorithm except that Algorithm 4 takes the new unfolded i-verifier V_u^p as input. Also, Algorithm 7 can return the empty automaton when $[\mathcal{L}_m(V_u^{e,trim})]^{\uparrow C} = \emptyset$.

Let us go back to V_u^p in Figure 4.13. The resulting AIS^p is the structure in Figure

Algorithm 7: Construct the AIS^p

```
input  : $V_u^p = (Y \cup Z, E_o \cup M_v, f_{yz} \cup f_{zy}, y_0)$
output: AIS$^p = (Y \cup Z, E_o \cup M_v, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$
1 Mark all the $Y$ states in $V_u^p$
2 Let $E_o$ be uncontrollable and $M_v$ be controllable
3 Trim $V_u^p$ and let $V_u^{e,trim}$ be the specification automaton
4 Obtain $[\mathcal{L}_m(V_u^{e,trim})]^{\uparrow C}$ w.r.t $\mathcal{L}(V_u^p)$ by following the $\uparrow C$ algorithm in [10]
5 Return the resulting automaton representation of $[\mathcal{L}_m(V_u^{e,trim})]^{\uparrow C}$, which is a
  sub-automaton of $V_u^{e,trim}$
```

4.13 without the shaded states. In this particular example, the structure of the AIS^p is the same as the AIS^e in Figure 4.9. However, the information states are different. We will show later in Section 4.8.1 an example where the structures of the AIS^p and the AIS^e are different.

Note that the AIS^p has the same properties as the AIS^e. That is, the AIS^p enumerates all i-enforcing insertion functions, and the AIS^p is not the empty automaton if and only if opacity is i-enforceable.

Remark IV.18. In the polynomial algorithm, we omit the language equality check after V in the exponential algorithm. That is, when $P_{und}[\mathcal{L}(V)] \neq P[L(G)]$, we will not stop but keep constructing the AIS^p. However, in this case, the V_u^p will always have blocking states as the insertion function cannot react to all the system output behavior. Consequently, $\uparrow C$ algorithm will return the empty automaton; i.e., the AIS^p is the empty automaton.

4.8.1 Correctness of the Construction

The polynomial algorithm differs from the exponential algorithm in how we obtain the unfolded i-verifier V_u. In this section, we will prove that constructing V_u^p using Algorithm 6 is correct. Specifically, we will prove in Lemma IV.19 and Lemma IV.20 that each Y and Z state in V_u^p is a correct *information state*, respectively. That is, each state contains enough information to enumerate all possible actions for its corresponding player.

Given that s has been output from the system and has been modified to $\tilde{s}$ by the insertion function, the following Y state in V_u^p is $y = m_v = \delta_v(m_{v,0}, \tilde{s})$. We now show that, using y, we can enumerate all possible next output events from G.

Lemma IV.19. *Consider that player G has output $s \in P[\mathcal{L}(G)]$ and player I has inserted s to $\tilde{s}$. The next turn of the game will be played by player G. State $m_v = \delta_v(m_{v,0}, \tilde{s})$ provides enough information to enumerate all possible next output events from player G.*

Proof. To determine all possible next output events from the system, it suffices to know the current state estimate of the system. Recall that $V := \mathcal{E}^d||_d\mathcal{E}^f$. By construction, transitions in $\mathcal{E}^f$ are only triggered by E_o and every $e_i \in E_i$ results in only a self-loop transition in $\mathcal{E}^f$. Hence, in $m_v = (m_d, m_f) = \delta_v(m_{v,0}, \tilde{s})$, we know that $m_f = f_{\mathcal{E}}(x_{\mathcal{E},0}, s)$ is current state estimate of the system, where $s = P_{und}(\tilde{s})$. □

Next, we show that, using $z \in Z$, which consists of its predecessor Y state and the event label of its incoming transition, we can enumerate all safe insertion choices for player I.

Lemma IV.20. *Consider that player G has output $se_o \in P[\mathcal{L}(G)]$ and player I has inserted s to $\tilde{s}$. The next turn of the game will be played by player I. The pair (m_v, e_o), where $m_v = \delta_v(m_{v,0}, \tilde{s})$, provides enough information to enumerate all possible safe insertion choices for e_o.*

Proof. Consider $se_o \in P[\mathcal{L}(G)]$ and $f_I^{str}(s) = \tilde{s}$, the set of all safe insertion choices is $\mathcal{I} = \{s_I \in E_i^* : \mathcal{M}_i(\tilde{s}s_I e_o) \in L_{safe}\}$. By construction, $\mathcal{L}(V) = \{t \in (E_o \cup E_i)^* : \mathcal{M}_i(t) \in L_{safe} \wedge P_{und}(t) \in P[\mathcal{L}(G)]\}$. Hence, $\mathcal{I} = \{s_I \in E_i^* : \tilde{s}s_I e_o \in \mathcal{L}(V)\} = \{s_I \in E_i^* : m_v = \delta_v(m_{v,0}, \tilde{s}) \wedge (\exists m_v' \in M_v \text{ s.t. } m_v' = \delta_v(m_v, s_I)) \wedge \delta_v(m_v', e_o) \text{ is defined}\}$. One can compute $\mathcal{I}$ by using (m_v, e_o), where $m_v = \delta_v(m_{v,0}, \tilde{s})$. Therefore, knowing (m_v, e_o) is enough to enumerate all possible safe insertion choices. □

As we have done in the exponential algorithm, we partition $\mathcal{I}$ into a finite number of sets using function $Ins()$ and enumerate the partition instead of enumerating all strings in $\mathcal{I}$. Specifically, $\mathcal{I} = \{Ins(m_v, m'_v) : m_v = \delta_v(m_{v,0}, \tilde{s}) \wedge (\exists s_I \in E_i^* \text{ s.t. } m'_v = \delta_v(m_v, s_I) \text{ is defined}) \wedge \delta_v(m'_v, e_o) \text{ is defined}\}$. To construct a finite game structure that enumerates all safe insertion choices, we use m'_v to represent the set of insertions $Ins(m_v, m'_v)$ for m_v.

Theorem IV.21. *The AIS^p enumerates all and only i-enforcing insertion functions. Opacity is i-enforceable if and only if the AIS^p is not the empty automaton.*

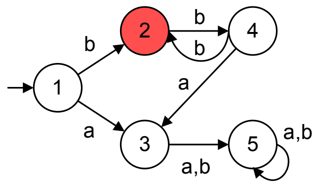
Proof. The proof follows from Lemma IV.19, Lemma IV.20, Theorem IV.13 and Theorem IV.14. □

4.8.2 Complexity of the Construction Procedure

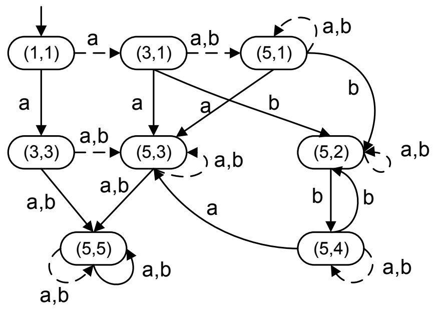
We use current-state opacity to analyze the computational complexity for the AIS^p. Given system G with $|X|$ states, the current-state estimator has at most $|X_\mathcal{E}| = 2^{|X|}$ states. To build the i-verifier V, we dashed parallel compose two special current-state estimators. Hence, V has at most $|M_v| = |X_\mathcal{E}|^2$ states. The unfolded i-verifier V_u^p, which uses M_v states to enumerate actions, has worst-case state space complexity $|X_{V_u^p}| = |M_v| + |E_o||M_v|$. Specifically, the first and the second terms account for the information states of the system and the insertion function, respectively. The time complexity for V_u^p is, on the other hand, is $O(|M_v|^3) = O(|X_\mathcal{E}|^6)$ because Algorithm 6 requires computing the connectivity between states of V. Finally, the AIS^p has at most $|X_{V_u^p}|$ states because the AIS^p is the recognizer for $\mathcal{L}_m(V_u^{e,trim})^{\uparrow C}$. Hence, in all, the space complexity of the AIS^p in the polynomial algorithm is $O\left((|E_o|+1)|X_\mathcal{E}|^2\right)$, reduced from exponential as compared to the AIS^e.

We now show an example where the AIS^p is smaller than the AIS^p.

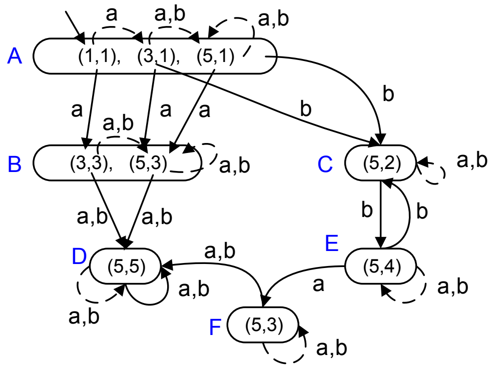
Example IV.22. Consider CSO of a system with $E_o = \{a, b, c\}$. We show directly the estimator $\mathcal{E}$ in Figure 4.14(a), whose states are the estimates of system states;



(a) Estimator $\mathcal{E}$ where state 2 is unsafe



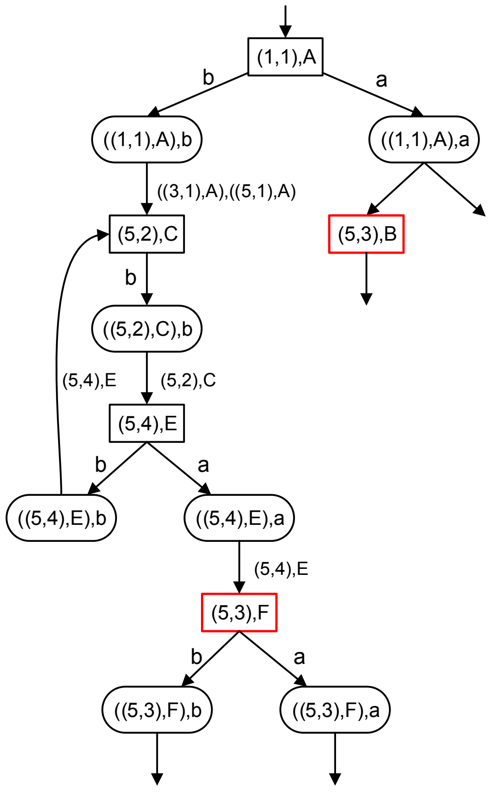
(b) V



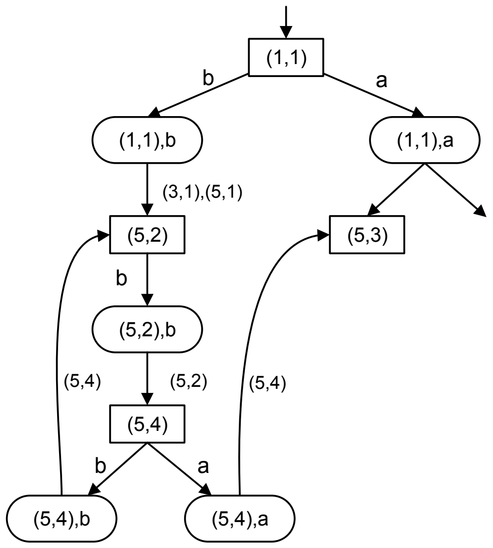
(c) $mObs(V)$

Figure 4.14: Relevant automata used in Example IV.22

for simplicity, these states are numbered from 1 to 5. Assume that state 2 is unsafe and that the other states are safe. In this example, we will build the AIS^e and AIS^p, and show how the AIS^e contains redundant information. We start with constructing the AIS^e. Shown in Figure 4.14(b) is V built from $\mathcal{E}$. We also build $mObs(V)$ in Figure 4.14(c). Then, we unfold the structure of $mObs(V)$, resulting in V_u^e that is shown in Figure 4.15(a). As can be seen in the Figure, no state is blocked. Hence, AIS^e equals V_u^e. On the other hand, to construct the AIS^p, we unfold directly the structure of V. The resulting V_u^p is shown in Figure 4.15(b), which is also the AIS^p. One can see that states $((5,3), B)$ and $((5,3), F)$ in the AIS^e are combined into state $(5,3)$ in the AIS^p. Although $((5,3), B)$ and $((5,3), F)$ have different observer states, their insertion choices are the same. This is because being in either observer state B or observer state F does not change the dashed connectivity from state $(5,3)$. In this example, the AIS^e has 20 states in total while AIS^p has only 17 states. The polynomial algorithm can reduce the state space complexity of the AIS.



(a) A partial structure of the AIS^e developed using the exponential algorithm (20 states in total)



(b) A partial structure of the AIS^p developed using the polynomial algorithm (17 states in total)

Figure 4.15: The AIS^e and the AIS^p in Example IV.22

Both the AIS^e and the AIS^p enumerate all i-enforcing insertion functions. Because the AIS^p has a smaller state space, hereafter, we will use AIS^p as our choice of the AIS. For simplicity, we will drop the superscript e in the AIS^e and simply call it the AIS.

4.9 Discussion: Opaque Communication

While the primary purpose of this chapter is to enforce opacity, it is useful to discuss the insertion enforcement mechanism in the context of *opaque communications*, as shown in Figure 4.16. Consider system G to be a sender that transmits messages $s \in P[\mathcal{L}(G)]$ through a public communications channel. Some messages are secret (corresponding to unsafe strings that reveal the secret); they are only for an intended receiver. Other messages are non-secret (corresponding to safe strings) and can be sent to any receiver. The communications channel is public. An intruder can easily eavesdrop on the transmitted message. To protect the secret messages, the system decides to "pack" each secret message $s \in P[\mathcal{L}(G)] \setminus L_{safe}$ in a non-secret-looking message s', where $\mathcal{M}_i(s') \in L_{safe}$. The "packing" is done by inserting additional internet packets that are represented by events in E_i. The resulting transmitted message is an innocuous message that contains hidden secret information. This idea is consistent with the technique of *Internet Steganography* in computer science [27], where secret messages are transmitted through a public channel with some hiding techniques. A typical technique embeds secret information in unused bits of packet headers, such as the IP headers Type of Service (TOS) field proposed in [27]. The usage of such bits are assumed to be communicated between the sender and receiver prior to the use of the stegosystem and thus is not known by others. Similarly, in opaque communications, we can label our inserted packet with an unused bit in the packet header. As discussed, the insertion label is shared only by G and the intended receiver. Thus, the intended receiver can reconstruct the secret message by applying projection P_{und} to the message. However, because the intruder cannot see the insertion label, it reads the message with mask $\mathcal{M}_i$ and believes all message are regular if the insertion function is i-enforcing. Consequently, our technique for opacity enforcement can be applied in the above-mentioned opaque communications model provided a mechanism for exchanging the insertion label is available.

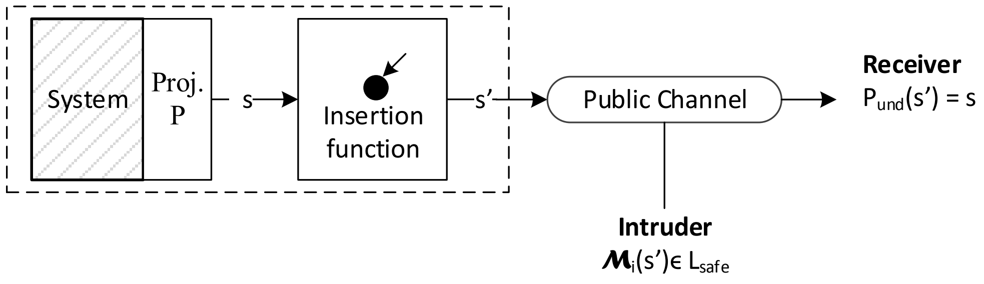


Figure 4.16: The opaque communication

4.10 Discussion: Intruder's Knowledge of f_I

So far, we have assumed that the intruder knows the structure of G but does not know how f_I is defined. By using an i-enforcing f_I, we assure that the intruder, not knowing f_I at the outset, would never figure out the existence of an insertion function. However, an interesting question that arises is what happens if the intruder knows f_I. In this section, two attack models are considered: (AM1) the intruder knows only G; and (AM2) the intruder knows G and f_I. We discuss how such knowledge of f_I affects the construction of an insertion function.

In both models (AM1) and (AM2), the intruder does not observe the insertion label, i.e., it observes through $\mathcal{M}_i \circ P_{oi}$. In (AM2), by knowing f_I, the intruder can construct the modified system $\tilde{G}$ and the modified secret and non-secret behaviors, i.e., $\tilde{L_S} := P_{oi}^{-1}(f_I^{str}[P(L_S)]) \cap \mathcal{L}(\tilde{G})$ and $\tilde{L_{NS}} := P_{oi}^{-1}(f_I^{str}[P(L_{NS})]) \cap \mathcal{L}(\tilde{G})$. Hence, checking if f_I enforces opacity within (AM2) is equivalent to verifying if $\tilde{G}$ is opaque with respect to $\tilde{L_S}, \tilde{L_{NS}}$, and $\mathcal{M}_i \circ P_{oi}$. That is, f_I enforces G to be opaque if and only if

$$\mathcal{M}_i\left(P_{oi}[\mathcal{L}(\tilde{G})]\right) \subseteq \mathcal{M}_i[P_{oi}(\tilde{L_{NS}})] \tag{4.3}$$

Since the intruder knows f_I, there is no need to keep the modified output within L_{safe}. The insertion function can create any $\tilde{L_S}$ and $\tilde{L_{NS}}$ as long as Equation (4.3) holds.

One may think opacity enforcement within (AM1) is a superclass of that within (AM2). However, opacity enforcement for these two attack models may lead to incomparable solutions. A solution for one attack model may not be a solution for the other attack model. The reasons are the following: under (AM1), we need $\mathcal{M}_i(P_{oi}[\mathcal{L}(\tilde{G})]) \subseteq L_{safe}$ but we do not need Equation (4.3); on the other hand, under (AM2), Equation (4.3) is required but $\mathcal{M}_i(P_{oi}[\mathcal{L}(\tilde{G})]) \subseteq L_{safe}$ is not. For example, Figure 4.17(a) shows a current-state estimator where states 7 and 8 are unsafe and others are safe. Under (AM1), we have an i-enforcing insertion function defined as $f_I(\varepsilon, a) = d_i a, f_I(\varepsilon, b) = a_i b$ and $f_I(s, e_o) = e_o$ otherwise. However, such an f_I will reveal the secret under (AM2) because $abc \in P(L_S)$ is mapped to $d_i abc$ and observing $\mathcal{M}_i[P_{oi}(a_i b)] = ab$ implies the occurrence of string b, which reveals the secret. On the other hand, we show in Figure 4.17(b) another current-state estimator where only state 3 is unsafe. Under (AM2), we can insert a_i before b to make $a_i b$ and ab observationally equivalent under $\mathcal{M}_i \circ P_{oi}$. However, $\mathcal{M}_i(P_{oi}[\mathcal{L}(\tilde{G})]) \not\subseteq L_{safe}$. This results in a tricky situation under (AM1) where if ab has indeed occurred, the intruder will be in estimator state 3 and the secret will be revealed.

4.11 Conclusion

We have considered the problem of enforcing opacity using insertion functions at the interface between the system and the intruder. In this novel opacity-enforcement paradigm, the insertion function dynamically changes the system's observed behavior

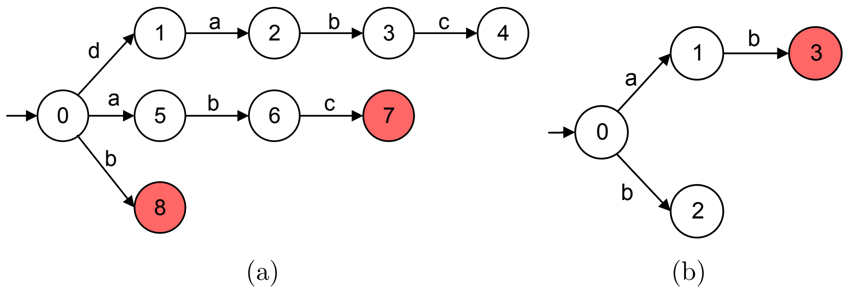


Figure 4.17: Current-state estimators used to show that (AM1) and (AM2) are not comparable

by inserting additional observable events. Such insertion functions must satisfy a property called "i-enforceability," which captures insertion functions' ability to force every system output behavior to be observationally equivalent to some non-secret behavior. To verify if a given insertion function is i-enforcing, we have constructed an automaton that captures the modified output behavior and used it to check if the modified behavior never reveals the secret. To verify if an opacity notion is i-enforceable, we have constructed the All Insertion Structure (AIS) that enumerates in a compact state structure all i-enforcing insertion functions. We have shown that opacity is i-enforceable if and only if the AIS is not the empty structure. Furthermore, using the AIS, we have presented an algorithm that synthesizes one i-enforcing insertion function. Lastly, the complexity of building the AIS was studied.

CHAPTER V

Synthesis of Optimal Insertion Functions for Opacity Enforcement

5.1 Introduction

In the previous chapter, we have studied the enforcement of opacity notions using insertion functions. Given a system that is not opaque, the so-called All Insertion Structure (AIS) is a bipartite graph that enumerates all valid insertion functions. Specifically, the AIS is a 2-player game structure that enumerates all system's output events at "system" states and all insertion choices at "insertion" states. We have provided an algorithm that synthesizes one random i-enforcing insertion function from the AIS. An interesting question that has not been addressed is how to synthesize an insertion function that is *optimal* in some specific sense.

In this chapter, we introduce the *maximum total cost* and the *maximum mean cost* to quantify insertion functions. The first cost captures the total insertion cost and the second cost considers the average insertion cost (per system output), both in the worst-case scenario. Specifically, we solve two optimization problems. We develop a test that determines if there is an insertion function that has a finite total cost. If such an insertion function exists, we then minimize the maximum total cost and synthesize an optimal *total-cost* insertion function. Otherwise, we minimize the maximum mean cost and synthesize an optimal *mean-cost* insertion function.

The synthesis of an optimal insertion function is solved by first finding an optimal strategy for the insertion function player on the AIS, and then using the optimal strategy to construct an *insertion automaton*. A strategy of the insertion function player is a mapping from every historical interaction of the system and the insertion function to an insertion action. It uniquely represents a given insertion function. An insertion automaton, on the other hand, is a compact encoding of an insertion function that can be easily composed with the system automaton. To find an optimal strategy, we leverage results from *minimax games* for the maximum total cost objective, and from *mean payoff games*, developed in [66], for the maximum mean cost objective. After the optimal strategy is found, we construct an insertion automaton, which is an I/O automaton that encodes the optimal insertion function. Our approach is inspired by [12] and [11], where an optimal dynamic observer is synthesized for fault diagnosis and opacity enforcement, respectively. But here we use insertion functions instead of dynamic observers.

The remaining sections of this chapter are organized as follows. Section 5.2 defines the maximum total cost and the maximum mean cost for insertion functions represented as insertion strategies on the AIS, and shows how to compute them. In Section 5.3, we consider the maximum total cost and present an algorithm for synthesizing an optimal total-cost insertion function. In Section 5.4, we consider the maximum mean cost and present an algorithm for synthesizing an optimal mean-cost insertion function. Finally, Section 5.5 concludes the chapter. Most of the results in this chapter also appear in [60].

5.2 Cost of An Insertion Function

To perform optimization on the AIS, we first define costs for all the insertion functions that are embedded in the transition structure of the AIS. Because our optimization procedures rely on finding optimal *insertion strategies* and since an insertion strategy uniquely defines an insertion function, we will define costs for insertion functions using insertion strategies.

Specifically, given a game, a player's strategy is a mapping from every game history where the player should move to an action. An *insertion strategy*, which is a strategy of player I, maps every path on the AIS that ends at a Z state to an outgoing edge of that Z state.

Definition V.1 (Insertion Strategy)**.** An insertion strategy on the AIS is a mapping $\pi : (E_o 2^{E_i^*})^* E_o \rightarrow 2^{E_i^*}$ that assigns an insertion action $L_i \in 2^{E_i^*}$ to every history where player I should play.

We can represent an insertion strategy as a (possibly infinite-state) bipartite graph $H = (Y_H \cup Z_H, E_o \cup 2^{E_i^*}, f_H, y_{H,0})$ where each Y_H state enumerates *all* system output events and each Z_H state selects *one* insertion action. In some cases, the bipartite graph representation H can be obtained by selecting outgoing actions for states of the AIS; such an insertion strategy is called *AIS-state-based*, or simply *state-based* hereafter. In general, we can obtain H by splitting the state space of the AIS as necessary using standard automata procedures. Only finite-state Hs are considered for our problem domains in this chapter. In the remainder of this section, we assume all Hs are finite. In Section 5.3, we will prove that there exists an optimal state-based strategy when an optimal one exists for the the considered cost objective.

5.2.1 The Weight Function w on the AIS

To define costs for insertion strategies, one needs to define a cost structure on the AIS. We begin with assigning a cost value to every inserted event. Cost function $c : E_i \rightarrow \{0, 1, 2, \ldots C_{max}\}$ maps each inserted event to a finite natural number. The domain of c is extended to E_i^* in a recursive additive manner by defining $c(\varepsilon) = 0$ and $c(se) = c(s) + c(e)$ where $s \in E_i^*, e \in E_i$. With function c, Definition V.2 that follows defines a weight function w on the transitions of the AIS. The weight value of a transition is the *minimum* insertion cost.

Definition V.2 (Weight Function w on the AIS)**.** Given the AIS $= (Y \cup Z, E_o \cup 2^{E_i^*}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$ and cost function c, we define weight function $w : (Y \times E_o \times Z) \cup (Z \times 2^{E_i^*} \times Y) \rightarrow \{0, 1, 2, \ldots W_{max}\}$ that maps each transition to its minimum insertion cost. Specifically, $w(y \xrightarrow{e_o} z) = 0$ and $w(z \xrightarrow{L_i} y) = \min\{c(s_I) : s_I \in L_i\}$ where $y \in Y, z \in Z, e_o \in E_o$ and $L_i \in 2^{E_i^*}$ is the set of inserted strings that labels the given transition.

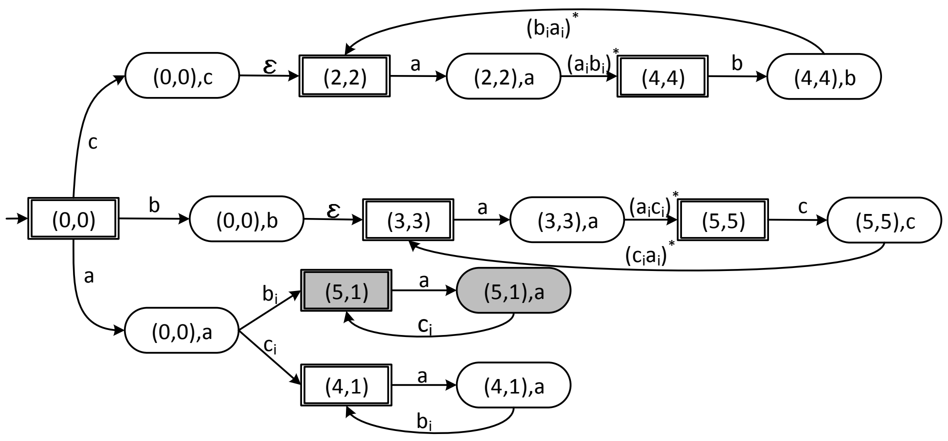
Note that transition $y \xrightarrow{e_o} z$ has weight zero as e_o is a system output event that contains no inserted event. For transition $z \xrightarrow{L_i} y$, we select from set L_i one string that achieves the minimum cost and assign that cost to the transition. This minimum is always well defined since there is at least one insertion that is bounded in length.

Example V.3. Consider the AIS in Figure 5.1(a). We calculate the weight function w with respect to cost function $c(a_i) = c(b_i) = 1, c(c_i) = 2$. Every $y \xrightarrow{e_o} z$ has a zero weight value. For $z \xrightarrow{L_i} y$, we find the minimum cost among all strings in L_i. Specifically, for transition $((2,2), a) \xrightarrow{(a_ib_i)^*} (4,4)$, we find the minimum insertion cost $c(\varepsilon) = 0$ and assign the transition weight to zero. Other transitions that are labeled with $(a_ib_i)^*$ or $(a_ic_i)^*$ are also assigned zero weight. For transitions labeled with b_i or c_i, we assign them weight 1 or 2, respectively. Finally, the AIS can be represented as the weighted graph in Figure 5.1(b), after the state names are relabeled by numbers and transition are labeled only by the edge weight.

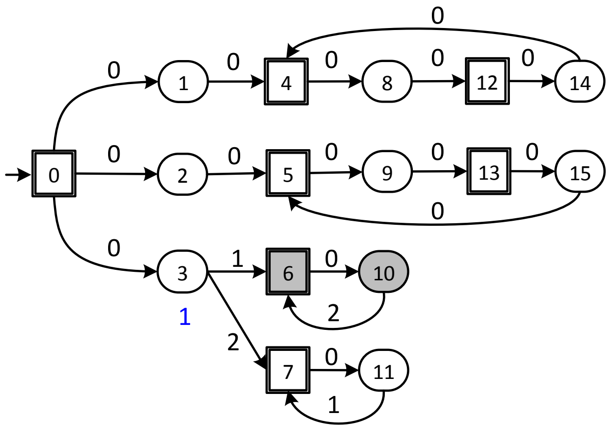
5.2.2 The Maximum Total Cost of An Insertion Strategy

We now extend the domain of weight function w to *paths* on the AIS and calculate the total cost of a path on the AIS.

Definition V.4 (Paths)**.** A path of k rounds that is generated by the AIS is a



(a) The AIS used in Example V.3



(b) The same AIS as in Figure 5.1(a) where information states are relabeled by numbers, transitions are labeled by the edge weights, and event labels are omitted. The blue number is the $\overline{c}^*(3)$ computed in Example V.22.

Figure 5.1: The AIS and its weighted graph with $c(a_i) = c(b_i) = 1, c(c_i) = 2$

sequence of transitions ending at a Y state: $p = y_0 \xrightarrow{e_1} z_0 \xrightarrow{L_1} y_1 \xrightarrow{e_2} z_1 \xrightarrow{L_2} \cdots y_{k-1} \xrightarrow{e_k} z_{k-1} \xrightarrow{L_k} y_k$, where $e_{i+1} \in E_o$, $L_{i+1} \in 2^{E_i^*}$, $z_i = f_{AIS,yz}(y_i, e_{i+1})$, and $y_{i+1} = f_{AIS,zy}(z_i, L_{i+1})$ for $0 \leq i \leq k-1$. The set of paths generated by the AIS is $\mathcal{P}aths(\text{AIS}) := \cup_{k\geq 0}\mathcal{P}ath_k(\text{AIS})$, where $\mathcal{P}ath_k(\text{AIS})$ is the set of all k-round paths.

Next, we define the total costs of paths.

Definition V.5 (Total Cost of A Path)**.** Consider path $p = y_0 \xrightarrow{e_1} z_0 \xrightarrow{L_1} y_1 \xrightarrow{e_2} z_1 \xrightarrow{L_2} \cdots y_{k-1} \xrightarrow{e_k} z_{k-1} \xrightarrow{L_k} y_k \in \mathcal{P}ath_k(\text{AIS})$. The total cost of p is $w(p) = \sum_{i=1}^{k}[w(e_i) + w(L_i)] = \sum_{i=1}^{k} w(L_i)$.

The last equality holds because $w(e_i) = 0,\ \forall e_i \in E_o$.

An insertion strategy has an infinite domain of paths in general; each path leads to a total cost. We consider the worst-case scenario by looking at the largest total cost. This cost is the *maximum total cost* of the insertion function that the insertion strategy encodes.

Definition V.6 (Maximum Total Cost of An Insertion Function)**.** Given insertion function f_I, its maximum total cost is $c_t(f_I) := c_t(H) := \limsup_{k\to\infty}\{\max\{w(p) : p \in S_k\}\}$, where $S_k := \{p : p \in \cup_{i=0}^{k}\mathcal{P}ath_i(\text{AIS})|_H)\}$ and H is the insertion strategy that uniquely defines f_I.

The notation $\mathcal{P}ath_k(\text{AIS})|_H$ refers to the restriction of H to k-round paths. We consider the limit superior because the system generates arbitrarily long strings in general and the total cost may not converge.

5.2.3 Calculation of the Maximum Total Cost

Algorithm 8 that follows computes the maximum total cost for an insertion function. Given insertion function f_I, we first construct insertion strategy H by selecting actions on the AIS according to f_I and splitting the state space of the AIS when needed. We then compute $c_t(f_I)$ directly on the structure of H. In the algorithm we denote by $n_H := |Y_H \cup Z_H|$ the cardinality of the state space of H.

Algorithm 8: The maximum total cost of f_I

```
  input  : Insertion function $f_I$ and cost function $c$
  output: $c_t(f_I)$
1 Encode $f_I$ as insertion strategy $H = (Y_H \cup Z_H, E_o \cup 2^{E_i^*}, f_H, y_{H,0})$
2 Find all strongly connected components $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ on $H$
3 for $C_i \in \mathcal{C}$ do
      if $\exists u \to u' \in C_i$ s.t. $w(u \to u') \neq 0$ then
          Return $\infty$
4 for $u \in Y_H \cup Z_H$ do
      Calculate $V_{|n_H|-1}(u)$ where
      $\forall u, V_k(u) := \max_{u \to u'}\{w(u \to u') + V_{k-1}(u')\}$ and $V_0(u) = 0$
5 Return $V_{|n_H|-1}(y_{H,0})$
```

The maximum total cost $c_t(f_I)$ is the maximum cost-to-go from the initial state of H. First, we determine if H contains any strongly-connected component (SCC) that has a non-zero edge weight. If there exists a non-zero-cost SCC (i.e., an SCC whose sum of all edge weights is non-zero), then there exists a path that loops in that SCC and incurs an infinite cost-to-go. In this case, we can immediately return $c_t(f_I) = \infty$. If, otherwise, there is no non-zero-cost SCC, then the maximum cost-to-go from the initial state equals the maximum *simple-path* cost-to-go from the initial state. Hence, we iteratively compute the $(|n_H| - 1)$-step cost-to-go from the initial state.

Denote by $|f_H|$ the number of transitions in H. In Algorithm 8, step 2 can be computed by using Tarjan's strongly connected components algorithm [57], which runs in $O(|n_H| + |f_H|)$. Step 4 can be computed in $O(|n_H||f_H|)$. Thus, $c_t(f_I)$ can be computed in $O(|n_H||f_H|)$.

Example V.7. Consider insertion function f_I encoded by the insertion automaton in Figure 5.2. We construct insertion strategy H that defines f_I by selecting in Figure 5.1(a) all states but the shaded states and selecting ϵ for transitions labeled by $(b_i a_i)^*, (a_i b_i)^*, (a_i c_i)^*$, or $(c_i a_i)^*$. Suppose the cost function is defined as $c(a_i) =$

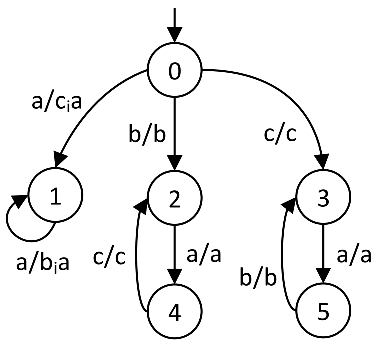


Figure 5.2: Insertion automaton IA

$1, c(b_i) = 0, c(c_i) = 2$. We calculate $c_t(f_I)$ by following Algorithm 8. First, we find all the SCC (*i.e.*, $\{(2,2), ((2,2),a), (4,4), ((4,4),b)\}, \{(3,3), ((3,3),a), (5,5), ((5,5),c)\}$, $\{(4,1), ((4,1),a)\}$) on H. Because the edge weights of all the SCCs are zero, we then iteratively compute $V_{|n_H|-1}(0) = V_{13}(0) = 2$. Thus, $c_t(f_I) = 2$. Now, let us change $c(b_i) = 1$. The edge weights of the AIS are shown in Figure 5.1(b). The SCC $\{(4,1), ((4,1),a)\}$ (or $\{7,11\}$ in Figure 5.1(b)) with the new cost function becomes a non-zero-cost self-loop. Hence, Algorithm 8 returns infinity.

5.2.4 The Maximum Mean Cost of An Insertion Strategy

Consider insertion function f_I for system G. The maximum total cost $c_t(f_I)$ always exists when G generates only strings of finite length. However, if G has a cycle, f_I may insert one or more events when G loops in that cycle, thereby resulting in an infinite $c_t(f_I)$. In this case, we compute the *maximum mean cost* of f_I, denoted by $\bar{c}(f_I)$, which considers the average insertion cost per system output event.

Definition V.8 (Maximum Mean Cost of An Insertion Function)**.** Given insertion function f_I, the maximum mean cost is $\bar{c}(f_I) := \bar{c}(H) := \limsup_{k\to\infty}\{\max\{\frac{1}{k}w(p) : p \in S_k\}\}$, where $S_k := \{p : p \in \cup_{i=0}^{k}\mathcal{P}ath_i(\text{AIS})|_H)\}$ and H is the insertion strategy that uniquely defines f_I.

The limit superior is taken as the maximum mean cost may not converge in the limit. If all paths on H are of finite length, then $\bar{c}(H) = 0$.

5.2.5 Calculation of the Mean Cost

We calculate $\bar{c}(f_I)$ on its insertion strategy H by treating H as a weighted graph using the given weight function. As seen in Definition V.8, the maximum mean cost is defined in terms of *rounds*. Since a round corresponds to two steps on H, the maximum mean cost of f_I is *double* of the "maximum mean weight" of H in the terminology of weighted graphs. We can calculate $\bar{c}(f_I)$ using the version of Karp's Theorem presented in [2] for *maximum* mean weight.[1]

Theorem V.9. *(Karp's Theorem [2]) Consider a weighted directed graph (X, f) with $|X| = n$, where X is the set of vertices and f is the set of edges. The maximum mean weight for a given initial vertex x_0 is,*

$$\lambda^* = \max_{x \in X} \min_{0 \le k \le n-1} \frac{F_n(x) - F_k(x)}{n-k} \tag{5.1}$$

where $F_k(x)$ is the maximum weight of an edge progression of length k from x_0 to x.

With the maximum mean weight of H defined, we now compute $\bar{c}(f_I)$ in Algorithm 9. The computation finishes in $O(|n_H| + |f_H|)$.

Algorithm 9: The maximum mean cost of f_I

```
input  : Insertion function $f_I$ and cost function $c$
output: $\bar{c}(f_I)$
1 Encode $f_I$ as insertion strategy $H = (Y_H \cup Z_H, E_o \cup 2^{E_i^*}, f_H, y_{H,0})$
2 Compute the maximum mean weight $\lambda^*$ of weighted graph $(Y_H \cup Z_H, f_H)$ using
  Equation (5.1)
3 Return $2\lambda^*$
```

Example V.10. Consider again the insertion function f_I in Figure 5.2. We construct insertion strategy H for f_I by removing the shaded states in Figure 5.1(a). Let the cost function be $c(a_i) = c(b_i) = 1, c(c_i) = 2$. We have shown in Example V.7 that,

[1]The original version of Karp's Theorem in [28] is for *minimum* mean weight.

with such a cost function, $c_t(f_I)$ goes to infinity. Here, we calculate the maximum mean cost using Algorithm 9. Consider the weighted graph $(Y_H \cup Z_H, f_H)$ of H, as shown in Figure 5.1(b) without the shaded states. The maximum mean weight is $\lambda^* = \frac{F_{14}(7)-F_{12}(7)}{14-12} = \frac{7-6}{2} = \frac{1}{2}$. Thus, $c_t(f_I) = 2\lambda^* = 1$. Insertion function f_I costs one per system output in the worst case.

5.3 Synthesis of An Optimal Finite-Cost Insertion Function

We have introduced the maximum total cost and the maximum mean cost for insertion functions. Given G that is not opaque, we want to find an optimal insertion function with respect to each cost. The two optimization problems are formulated as follows. In the problem statements, we use $H \in$ AIS to denote that insertion strategy H is obtained from the AIS after potential state splitting.

Problem V.11. *Consider G that is not opaque and cost function c for inserted events. Find:*
(a) the optimal maximum total cost $c_t^* = \min\{c_t(H) : H \in AIS\}$
(b) an optimal total-cost insertion function that achieves c_t^*

Problem V.12. *Consider G that is not opaque and cost function c for inserted events. Find:*
(a) the optimal maximum mean cost $\bar{c}^* = \min\{\bar{c}(H) : H \in AIS\}$
(b) an optimal mean-cost insertion function that achieves $\bar{c}^*$

Notice that if c_t^* is finite, there is no need to solve Problem V.12 as $\bar{c}^*$ is known to be zero and an optimal total-cost insertion function is an optimal mean-cost insertion function. Hence, our goal is to synthesize an optimal total-cost insertion function if c_t^* is finite, and an optimal mean-cost insertion function otherwise. Problem V.11 is solved in Sections 5.3.1 to 5.3.3 while Problem V.12 is solved in Section 5.4.

5.3.1 Minimax Game Formulation for An Optimal Total-Cost Insertion Function

Recall that the AIS is a game structure that enumerates, in alternate turns, the actions of the system player and those of the insertion function player. To solve Problem V.11, we consider a *minimax game* on the AIS and find an optimal insertion strategy. In the minimax game, the system player tries to maximize $\liminf_{k\to\infty}\sum_{i=1}^{k} w(u \xrightarrow{e} u')$ and the insertion function player tries to minimize $\limsup_{k\to\infty}\sum_{i=1}^{k} w(u \xrightarrow{e} u')$, where $u, u' \in Y \cup Z$ and $e \in E_o \cup 2^{E_i^*}$. Because the optimal maximum total cost c_t^* is defined in terms of the worst-case scenario, it is indeed the resulting $\limsup_{k\to\infty}\sum_{i=1}^{k} w(u \xrightarrow{e} u')$ in the minimax game. Moreover, the optimal insertion strategy is the resulting strategy that minimizes $\limsup_{k\to\infty}\sum_{i=1}^{k} w(u \xrightarrow{e} u')$. As we will show later, there is an optimal state-based insertion strategy, meaning that the strategy can be represented as a subgraph of the AIS. We will use the subgraph to construct an optimal insertion function in Section 5.3.3.

5.3.2 Finding the Optimal Total Cost

The system we consider generates arbitrarily long strings in general, and thus the game described by the AIS is infinite horizon. In this section, we solve Problem V.11 by solving a *finite-horizon minimax game* played on the AIS, where player P_1 maximizes the total cost at Y states and player P_2 minimizes the total cost at Z states, for a finite number of steps. Specifically, we use the optimal total cost for the finite-horizon minimax game to determine c_t^*, and then use c_t^* to find an optimal insertion strategy.

Consider the AIS $= (Y \cup Z, E_o \cup 2^{E_i^*}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$. Denote by $V_k(u)$ the optimal total cost and by $a_k(u)$ an optimal action in the k-step game assuming the game starts at state u, where $u \in Y \cup Z$. We calculate the cost and the action using the following recursive equations.

$$V_k(u) = \begin{cases} \max\limits_{(u \to u')} \{w(u \xrightarrow{e} u') + V_{k-1}(u')\}, \text{ if } u \in Y \\ \\ \min\limits_{(u \to u')} \{w(u \xrightarrow{e} u') + V_{k-1}(u')\}, \text{ if } u \in Z \end{cases} \tag{5.2}$$

where $V_0(u) = 0$ for $u \in Y \cup Z$.

$$a_k(u) = \begin{cases} \arg\max\limits_{(u \to u')} \{w(u \xrightarrow{e} u') + V_{k-1}(u')\}, \text{ if } u \in Y \\ \\ \arg\min\limits_{(u \to u')} \{w(u \xrightarrow{e} u') + V_{k-1}(u')\}, \text{ if } u \in Z \end{cases} \tag{5.3}$$

Note that the strategy found using Equation (5.3) is not state-based in general, as the optimal action depends also on k.

It turns out that finite-horizon minimax games can be used to analyze the *infinite-horizon minimax game.* Denote by $V(y_0)$ the optimal total cost and by π^* an optimal strategy for P_1 and P_2 in the infinite-horizon minimax game.[2] Let n be the number of states in the AIS. We will prove in Theorem V.15 that $V(y_0)$ and a state-based π^* can be found in the n^2W_{max}-step game. In the following, we let $V_k^{\pi}(y_0)$ be the total cost in the k-step game when the game starts at y_0 and strategy π is used, and let π_k^* be an optimal strategy in the k-step game.

Lemma V.13. *Let $l = n^2W_{max}$. If $V_l(y_0) < nW_{max}$, then there exists a state-based optimal strategy π_l for the l-step game such that $V_{l'}^{\pi_l}(y_0) = V_l^{\pi_l}(y_0), \forall l' \geq l$.*

Proof. Consider outcome path p_l on the AIS, resulting from the players playing π_l^*. Label actions on p_l by $a_1a_2 \ldots a_l$. We will first construct a state-based strategy π_l that is as good as π_l^*. Then, we show that if π_l is used for l'-step game ($\forall l' \geq l$), the optimal total cost would be the same as that for the l-step game.

[2]Only one player will move at a given history of the game. Thus, we can treat the strategies of the two players as a single strategy.

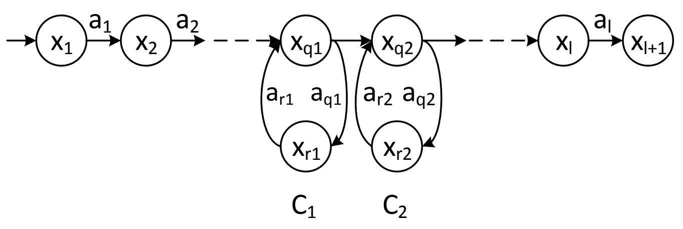


Figure 5.3: An outcome path used to illustrate the proofs of Lemmas V.13 and V.14

Partition p_l into $a_1 \dots (a_{q_1} \dots a_{r_1}) \dots (a_{q_2} \dots a_{r_2}) \dots (a_{q_{N_c}} \dots a_{r_{N_c}}) \dots a_l$, where $(a_{q_i} \dots a_{r_i})$ is the i-th cycle C_i on p_l, as shown in Figure 5.3. Because a cycle is formed within at most n steps, $N_c \geq l/n = n^2 W_{max}/n = nW_{max}$. First, we argue that each of these cycles has a zero cycle cost. Suppose there exists a non-zero cost cycle C_{nz} (say C_1). If x_{q_1} is played by P_1, then P_1 has a better strategy for which the outcome path reaches C_{nz} and loops there until step l. In this case, the number of times in C_{nz} must be greater than N_c and the path would incur a total cost greater than nW_{max}. This contradicts the hypothesis in the statement of the lemma that $V_l(y_0) < nW_{max}$ and that p_l is the optimal outcome path. On the other hand, if x_{q_1} is played by P_2, then P_2 has a better strategy for which the outcome path skips C_{nz} and loops in a zero-cost cycle for the extra steps. This also contradicts the hypothesis that p_l is the optimal outcome path. Hence, all the cycles have zero-cost. Then, we argue that actions $a_{r_{N_c}+1}, \dots, a_l$ are also zero-cost using a similar reasoning. Suppose any of them has a non-zero cost. Then, P_1 has a better strategy for which the outcome path skips C_1 to C_{N_c} and loops in a cycle containing $a_{r_{N_c}+1}, \dots, a_l$. Also, P_2 has a better strategy that loops in zero-cost cycle C_{N_c} until step l. Either case contradicts the hypothesis that p_l is the optimal outcome path. Hence, $a_{r_{N_c}+1}, \dots, a_l$ are zero-cost.

Now, let us construct a new path p'_l from p_l as follows. First, find on p_l the smallest i such that all actions after a_{r_i} have a zero cost. Then, remove cycles $C_1, \dots C_{i-1}$ and all actions after a_{r_i}. The resulting p'_l is $a_1 \dots a_{q_1-1} a_{r_1+1} \dots a_{q_2-1} a_{r_2+1} \dots (a_{q_i} \dots a_{r_i})$, which ends in cycle C_i. Define state-based strategy π_l by assigning to each state the only outgoing action according to p'_l. Use π_l as the strategy for the l-step game. The resulting outcome path would begin with p'_l as a subpath and then loop in C_i until step l. Because this path differs in p_l only in the replacement of some zero-cost actions, the corresponding total cost is the same as that for p_l. That is, $V_l^{\pi_l}(y_0) = V_l(y_0)$. Therefore, π_l is an optimal strategy for the l-step game.

Finally, because C_i is a zero-cost cycle, using π_l for any l'-step game ($l' \geq l$) would not increase the total cost as the outcome path will cycle in C_i. Hence, $V_{l'}^{\pi_l}(y_0) = V_l^{\pi_l}(y_0), \forall l' \geq l$. □

Lemma V.14. *Let $l = n^2 W_{max}$. If $V_l(y_0) \geq nW_{max}$, then $V_{k+n}(y_0) > V_k(y_0), \forall k \geq 0$.*

Proof. Consider the same setups used in Lemma V.13. We partition p_l again into $a_1 \ldots (a_{q_1} \ldots a_{r_1}) \ldots (a_{q_2} \ldots a_{r_2}) \ldots (a_{q_{N_c}} \ldots a_{r_{N_c}}) \ldots a_l$. This time, it is hypothesized that $V_l(y_0) \geq nW_{max}$. We will argue that all the cycle costs are non-zero.

If all the cycles are zero cost, then $V_l(y_0) < nW_{max}$ because at most $n-1$ steps do not belong to a cycle. But this violates the hypothesis of the lemma. Suppose there is a zero-cost cycle C_z (say C_2). Then, if x_{q_2} is played by P_1, then P_1 can do better by skipping C_z and using the extra steps on a non-zero cost cycle. On the other hand, if x_{q_2} is played by P_2, then P_2 can do better by staying in C_z until step l. Both cases contradict our assumption that p_l is an optimal outcome path. Therefore, all the cycle costs on p_l must be non-zero.

Consider the k-step game for a given $k \geq 0$ and let p_k be an optimal outcome path for the k-step game. If $k < l$, all the cycle costs on p_k must be non-zero. Otherwise, for P_1, using the first k steps of p_l results in a better strategy; for P_2, using p_k for the first k step of the l-step game results in a path better than p_l. If $k > l$, all the cycles on p_k must be non-zero cost as well. Otherwise, if there exists a zero-cost cycle, then P_1 could perform at least as well by skipping that cycle and using the extra steps on a non-zero cost cycle; P_2 would prefer to stay in that zero-cost cycle. Also, if all the cycles are zero-cost, then $V_k(y_0) < nW_{max}$, which would lead to the wrong conclusion that $V_k(y_0) < V_l(y_0)$. Therefore, if $V_l(y_0) \geq nW_{max}$, then all the cycles on p_k are non-zero cost, $\forall k \geq 0$.

Now, let us compare $V_{k+n}(y_0)$ and $V_k(y_0)$. We partition p_k and p_{k+n} into segments of cycles, as we have done for p_l. Because a cycle is formed within n steps, p_{k+n} must have at least one more cycle than p_k. Since all the cycles are non-zero cost, $V_{k+n}(y_0) > V_k(y_0)$. □

Theorem V.15. *Let $l = n^2 W_{max}$. If $V_l(y_0) < nW_{max}$, then $V(y_0) = V_l(y_0)$ and is achieved by a state-based optimal strategy π_l for the l-step game. Otherwise, $V(y_0)$ goes to infinity.*

Proof. When $V_l(y_0) < nW_{max}$, we have constructed in Lemma V.13 a state-based optimal strategy π_l for the l-step game. Use π_l for the infinite game. We have $\lim_{l'\to\infty} V_{l'}^{\pi_l}(y_0) = V_l(y_0) \leq \lim_{l'\to\infty} V_{l'}^{\pi^*}(y_0)$. The first equality is according to Lemma V.13; the second inequality is because π^* is an optimal strategy for the infinite game.

Now, use π^* for the l-step game. Because π_l is the optimal strategy for the l-step game, we have $V_l^{\pi^*}(y_0) \leq V_l(y_0) < nW_{max}$. Partition the optimal outcome path p_∞ for the infinite game into $a_1 \ldots (a_{q_1} \ldots a_{r_1}) \ldots (a_{q_2} \ldots a_{r_2}) \ldots$, where $(a_{q_i} \ldots a_{r_i})$ is the i-the cycle C_i on p_∞. Because p_∞ is an optimal outcome path, the cycle costs must be either all zero or all non-zero. Otherwise, it would not be optimal, by the same argument as in the proofs of Lemmas V.13 and V.14. Since $V_l^{\pi^*}(y_0) < nW_{max}$, all cycles in the first l steps must be zero-cost according to Lemma V.13. Thus, all cycle costs on p_∞ are zero and there are at most $n-1$ non-zero cost edges on p_∞. Now, let us construct p'_∞ from p_∞ by skipping cycles until the last non-zero-cost edge and then looping in a zero-cost cycle for the extra steps. The total cost of p'_∞ is the sum of the first $n-1$ steps, which equals that of p_∞. That is, p'_∞ corresponds to another optimal strategy π'^* for the infinite game. Consequently, we have $V(y_0) = \lim_{l'\to\infty} V_{l'}^{\pi'^*}(y_0) = V_{n-1}^{\pi'^*}(y_0) = V_l^{\pi'^*}(y_0) \leq V_l^{\pi_l}(y_0) = \lim_{l'\to\infty} V_{l'}^{\pi_l}(y_0)$.

Finally, combine the above two inequalities. We obtain $V(y_0) := \lim_{l'\to\infty} V_{l'}^{\pi'^*}(y_0) = \lim_{l'\to\infty} V_{l'}^{\pi_l}(y_0) = V_l^{\pi_l}(y_0) = V_l(y_0)$; state-based strategy π_l is optimal for the infinite game and $V(y_0) = V_l(y_0)$.

On the other hand, when $V_l(y_0) \geq nW_{max}$, we have $V_{k+n}(y_0) > V_k(y_0), \forall k \geq 0$ by Lemma V.14. Take k to infinity; the optimal total cost for the infinite game $V(y_0)$ goes to infinity. □

We have proven in Theorem V.15 that $c_t^* = V(y_0)$ can be calculated in a finite-horizon minimax game. Hence, we now combine all the above results and compute c_t^* in Algorithm 10. Notice that if the AIS is acyclic, then $V(y_0)$ will converge within n steps and $V(y_0) < nW_{max}$; that is, $c_t^* < \infty$.

Algorithm 10: The optimal total cost c_t^*

```
input  : AIS = $(Y \cup Z, E_o \cup 2^{E_i^*}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$ and weight function $w$
output: $c_t^*$
1 Compute $V_l(y_0)$ for $l = n^2 W_{max}$ using Equation (5.2)
2 if $V_l(y_0) < nW_{max}$ then
      Return $V_l(y_0)$
  else
      Return $\infty$
```

Example V.16. Let us consider the AIS in Figure 5.1(a) with cost function $c(a_i) = 1$, $c(b_i) = 0$, $c(c_i) = 2$. We compute c_t^* by following Algorithm 10. In step 1, $V_l((0,0)) = 2$, where $l = n^2 W_{max} = 16^2 \cdot 2 = 512$. Because $V_l((0,0)) < nW_{max} = 32$, we have $c_t^* = V_l((0,0)) = 2$. Now, if we change $c(b_i) = 1$, then $V_l((0,0)) = 257 > nW_{max} = 32$. Therefore, c_t^* goes to infinity in this case.

Corollary V.17. *There exists optimal state-based total-cost insertion functions if and only if* $V_l(y_0) < nW_{max}$.

Proof. The proof follows directly from Theorem V.15. □

5.3.3 Synthesis of the Optimal Total-Cost Insertion Function

Algorithm 10 calculates c_t^*. If $c_t^* < \infty$, then there exists an optimal total-cost insertion function and we will synthesize one using Algorithm 11. Otherwise, we go to Section 5.4 and solve Problem V.12.

When $V_l(y_0) < nW_{max}$, by Corollary V.17, there is an optimal state-based strategy. In Algorithm 11 that follows, we build optimal state-based strategy H that selects all the actions at Y states and optimal actions at Z states from the AIS, resulting in a subgraph of the AIS. Note that Algorithm 11 computes an optimal insertion strategy in a breadth-first manner, thereby ignoring Z states that are never reached.

```
Algorithm 11: Find an optimal total-cost insertion strategy
  input : AIS= $(Y \cup Z, E_o \cup 2^{E_i^*}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$ and weight function $w$
  output: Optimal strategy $H = (Y_H \cup Z_H, E_o \cup 2^{E_i^*}, f_H, y_{H,0})$
1 for $u \in Y \cup Z$ do
      Compute $a_l(u)$ for $l = n^2 W_{max}$ using Equation (5.3)
2 Let $y_{H,0} = y_0$. Set $Y_H = \{y_{H,0}\}$
3 for $u \in Y_H$ that has not been examined do
      for $e \in E_o$ do
          $f_H(u,e) := f_{AIS,yz}(u,e)$ if $f_{AIS,yz}(u,e)$ is defined
4 for $u \in Z_H$ that has not been examined do
      $f_H(u,e) := f_{AIS,yz}(u,e)$ where $e = a_l(u)$ is an optimal action for $u$
5 Go back to step 2 until all selected states have been examined
```

Once we obtain optimal state-based insertion strategy H from Algorithm 11, we build an insertion function from H. Without loss of generality, the insertion function is encoded as an insertion automaton, using Algorithm 12.

```
Algorithm 12: Construct an insertion automaton from an insertion strategy
  input : $H = (Y_H \cup Z_H, E_o \cup 2^{E_i^*}, f_H, y_{H,0})$, and weight function $w$
  output: IA= $(X_{ia}, E_o, E_i^* E_o, f_{ia}, q_{ia}, x_{ia,0})$
1 Let $x_{ia,0} = y_{H,0}$, $X_{ia} = \{x_{ia,0}\}$
2 for $x \in X_{ia}$ that has not been examined do
      for $x \xrightarrow{e_o} z \xrightarrow{L_i} y$ where $x, y \in Y_H, z \in Z_H$ do
          Add $y$ to $X_{ia}$
          Find $s_I \in L_i$ that corresponds to $w(z \xrightarrow{L_i} y)$
          Define $f_{ia}(x, e_o) = y$ and $q_{ia}(x, e_o) = s_I e_o$
3 Go back to step 2 until all states in $X_{ia}$ have been examined
```

Example V.18. Consider again the AIS in Figure 5.1(a) with cost function $c(a_i) = 1,\ c(b_i) = 0,\ c(c_i) = 2$. We have computed $c_t^* = 2$ in Example V.16 and concluded that an optimal total-cost insertion function exists. In this example, we want to synthesize an optimal total-cost insertion function. First, we apply Algorithm 11 to obtain an optimal strategy H. Specifically, we select all outgoing actions for square-shaped states, action $((0,0),a) \xrightarrow{c_i} (4,1)$ for state $((0,0),a)$, and the only actions for the other ellipse-shaped states. Such a selection results in an optimal strategy H that is the AIS in Figure 5.1(a) without the shaded states. Then, we apply Algorithm 12 by taking H as input to construct an insertion automaton. For each $y \xrightarrow{e_o} z \xrightarrow{L_i} y'$ where $y, y' \in Y|_H, z \in Z|_H$, we first find the inserted string s_I from Figure 5.1(a) as follows: select ε for transitions labelled with $(a_ib_i)^*$ or $(b_ia_i)^*$; select the only string for the other transitions. Then, using the chosen s_I, we redefine the transition to be $y \xrightarrow{e_o/s_Ie_o} y'$. The resulting optimal insertion automaton is shown in Figure 5.4, with state names relabeled according to Figure 5.1(b).

Theorem V.19. *Applying Algorithms 10, 11 and 12 solves Problem V.11(b).*

Proof. Algorithm 10 follows from Theorem V.15. In Algorithm 11, an insertion action is chosen for every system output event. Hence, the resulting insertion strategy is i-enforcing. The strategy is optimal as all insertion actions are optimized. Since the insertion automaton in Algorithm 12 is constructed from the optimal strategy, it encodes an insertion function that achieves c_t^*. □

In all, given the AIS$= (Y \cup Z, E_o \cup 2^{E_i^*}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$, computing an optimal total-cost insertion function can be done in $O(n^2|f_{AIS}|W_{max})$, where n is the number of states in the AIS and $|f_{AIS}|$ is the number of transitions in $f_{AIS} = f_{AIS,yz} \cup f_{AIS,zy}$.

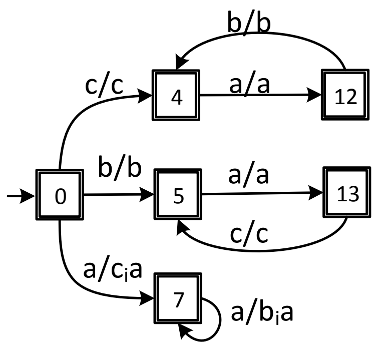


Figure 5.4: The optimal IA in Examples V.18 and V.24, where the state names are relabeled according to Figure 5.1(b).

5.4 Synthesis of An Optimal Mean-Cost Insertion Strategy

5.4.1 Mean Payoff Game Formulation of the Synthesis Problem

To solve Problem V.12, we solve a *mean payoff game* on the AIS. Similarly to Section 5.3, we let the AIS be our game structure. Here, the insertion function player tries to minimize $\limsup_{k\to\infty}\frac{1}{k}\sum_{i=1}^{k} w(u \xrightarrow{e} u')$ and the system player tries to maximize $\liminf_{k\to\infty}\frac{1}{k}\sum_{i=1}^{k} w(u \xrightarrow{e} u')$ where $u, u' \in Y \cup Z$ and $e \in E_o \cup 2^{E_i^*}$. The optimal maximum mean cost $\bar{c}^*$ is *double* of the resulting $\limsup_{k\to\infty}\frac{1}{k}\sum_{i=1}^{k} w(u \xrightarrow{e} u')$ because $\bar{c}^*$ is the worst-case average cost per *round*. Also, the optimal insertion strategy is the resulting strategy that minimizes $\limsup_{k\to\infty}\frac{1}{k}\sum_{i=1}^{k} w(u \xrightarrow{e} u')$. In a mean payoff game, both players have state-based optimal strategies [22]. Hence, the resulting optimal strategy will be a subgraph H of the AIS that selects all the actions of the system at Y states but only *one* optimal action at Z states. In the following, we first find the optimal mean cost $\bar{c}^*$ in Section 5.4.2, and then synthesize the optimal insertion function that achieves $\bar{c}^*$ in Section 5.4.3.

5.4.2 Finding the Optimal Mean Cost

We begin with solving Problem V.12(a) and finding $\bar{c}^*$. This problem is a special instance of the problem of calculating the *value* of the mean payoff game on weighted automata that is defined in [22]. Hence, we can compute $\overline{c}^*$ by adapting the results in [22]. Specifically, here, the AIS is our weighted automaton. Computing $\overline{c}^*$ on the AIS differs from computing the game value on a general weighted graph in the following aspects: (1) the game value is the average cost *per step in the game*, whereas $\overline{c}^*$ is the average cost *per round* (i.e., per system output event); (2) all edge weights in the AIS are non-negative while edge weights in [66] can be negative; and (3) every edge from a Y state of the AIS has a zero weight while edge weights in [66] are non-zero in general. Let us denote by $\bar{V}(u)$ the game value assuming that the game starts from state u. To find $\overline{c}^*$ on the AIS, we first address differences (2) and (3) by establishing in Theorem V.20 a tighter bound for $\bar{V}(u)$, where u is a state of the AIS. This bound allows us to determine the correct value for $\bar{V}(u)$. Then, we address difference (1) by doubling the value of $\bar{V}(y_0)$ to obtain $\overline{c}^*$ in Algorithm 13.

Theorem V.20. *For every state u of the AIS, we have*

$$\frac{V_k(u)}{k} - \frac{n-1}{2k}W_{max} \leq \bar{V}(u) \leq \frac{V_k(u)}{k} + \frac{n-1}{2k}W_{max}$$

Proof. The proof follows the reasoning in the proofs of Theorems 2.2 and 2.3 in [66] but it is adapted for the special cost structure of the AIS. Consider a k-step game and the outcome path that is resulted from players playing π_k^*. The resulting k-step cost-to-go from state u on the outcome path is $V_k(u)$. We have $V_k(u) \leq k\bar{V}(u) + \lceil \frac{n-1}{2} \rceil W_{max}$. The first term on the right-hand side is because at most k steps are in a cycle, and if P_2 plays according to its optimal strategy, then the average cost of that cycle is at most $\bar{V}(u)$. The second terms on the right-hand side is because there can be at most $\lceil \frac{n-1}{2} \rceil$ non-zero steps before the cycle and each of them is at most W_{max}. Similarly, consider that P_1 plays according to its optimal strategy. We have $V_k(u) \geq (k-(n-1))\bar{V}(u) + 0$, as there are at least $k-(n-1)$ steps in the cycle and there can be zero steps before the cycle. Because $\bar{V}(u) \leq \frac{1}{2}W_{max}$, this inequality implies $V_k(u) \geq k\bar{V}(u) - \frac{n-1}{2}W_{max}$. By rearranging the above two inequalities, we have $\frac{V_k(u)}{k} - \frac{n-1}{2k}W_{max} \leq \bar{V}(u) \leq \frac{V_k(u)}{k} + \frac{n-1}{2k}W_{max}$. $\square$

With Theorem V.20, we can obtain a tighter bound for $\bar{V}(u)$ by considering a larger k. Because each edge is assumed to have an integer cost value and that a cycle is formed within at most n steps, $\bar{V}(u)$ is a rational number with a denominator at most n. Hence, the minimum distance between two possible values of $\bar{V}(u)$ is $\frac{1}{n(n-1)}$. Now, let us choose $k = n^3W_{max}$. The value of $\bar{V}(u)$ is then bounded in $\frac{V_k(u)}{k} - \frac{1}{2n(n-1)} < \frac{V_k(u)}{k} - \frac{n-1}{2n^3} \leq \bar{V}(u) \leq \frac{V_k(u)}{k} + \frac{n-1}{2n^3} < \frac{V_k(u)}{k} + \frac{1}{2n(n-1)}$ where only one valid value exists. Therefore, we can determine $\bar{V}(u)$ by searching within $[\frac{V_h(y_0)}{h} - \frac{1}{2n(n-1)}, \frac{V_h(y_0)}{h} + \frac{1}{2n(n-1)}]$ for $h = n^3W_{max}$.

Finally, we find the optimal mean cost by doubling the value of $\bar{V}(y_0)$. The whole process is captured in Algorithm 13.

```
Algorithm 13: Find the optimal mean cost for state $u$
  input : AIS= $(Y \cup Z, E_o \cup 2^{E_i^*}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$ and the weight function $w$
          and state $u \in Y \cup Z$
  output: $\overline{c}^*(u)$
1 Compute $V_h(u)$ for $h = n^3W_{max}$ using Equation (5.2)
2 Compute the $h$-step mean cost $V_h(u)/h$
3 Find the only rational number $r$ with a denominator at most $n$ that lies in the
  interval $[V_h(u)/h - \alpha,, V_h(u)/h + \alpha]$ with $\alpha = \frac{1}{2n(n-1)}$
4 Return $2r$
```

In Algorithm 13, $\overline{c}^*(u)$ is the optimal mean cost assuming that the game begins at state u. When $u = y_0$, the returned $\overline{c}^*(y_0)$ is the optimal mean cost $\overline{c}^*$. Note that we can compute $V_h(y_0)$ by continuing the computation of $V_l(y_0)$ in Algorithm 10, as $h = n^3W_{max}$ is greater than $l = n^2W_{max}$. In the next section, we will use $\overline{c}^*(u)$ to compute the optimal insertion function.

Example V.21. Consider again the weighted graph in Figure 5.1(b). We have shown in Example V.16 that no optimal total-cost insertion function exists when the cost structure is $c(a_i) = 1$, $c(b_i) = 1$, $c(c_i) = 2$. Here, we will synthesize an optimal mean-cost insertion function. In this example, we first calculate the optimal mean cost by following Algorithm 13. Then, we will finish the synthesis in Examples V.22 and V.24. In step 1, $V_h(0) = 4097$ where $h = n^3 W_{max} = 16^3 \cdot 2 = 8192$. Dividing $V_h(0)$ by h, we obtain the h-step mean cost $V_h(0)/h = 0.50012$. Finally, searching within the interval $[\frac{4097}{8192} - \frac{1}{480}, \frac{4097}{8192} + \frac{1}{480}] = [0.498, 0.502]$, we find that $\frac{1}{2}$ is the only valid value. The optimal mean cost is $\overline{c}^* = 2 \cdot \frac{1}{2} = 1$.

5.4.3 Synthesis of the Optimal Infinite-Cost Insertion Function

With Algorithm 13 that solves $\overline{c}^*(u), \forall u \in Y \cup Z$, at hand, we now find an optimal action for a given Z state using Algorithm 14. This algorithm, adapted from [66], eliminates insertion actions using a binary search technique. Notice that Algorithm 14 is applied only to insertion states, i.e., Z states. For every state $z \in Z$, denote by $d(z)$ the number of outgoing actions at z. By construction, we have $d(z) \geq 1$ because there are no deadlocked states in AIS. Therefore, the algorithm always outputs a valid action when it terminates.

Algorithm 14: Find the optimal action for state $z \in Z$

```
input : AIS= $(Y \cup Z, E_o \cup 2^{E_i^*}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$, weight function $w$, and a
        state $z \in Z$
output: Optimal action $L_i \in 2^{E_i^*}$
1 Compute $\overline{c}^*(z)$ by applying Algorithm 13
2 while $d(z) > 1$ do
      Remove $\lceil d(z)/2 \rceil$ outgoing actions at $z$ but leave at least one action
      Recompute the optimal cost, say $\overline{c}^*(z)'$, for the reduced AIS
      if $\overline{c}^*(z)' == \overline{c}^*(z)$ then
          The optimal action is one of the remaining actions at $z$
      else
          The optimal strategy is one of the removed actions at $z$
3 if $d(z) == 1$ then
      Return the only one action
```

Example V.22. We calculate the optimal action for state 3 in Figure 5.1(b) by following Algorithm 14. In step 1, we compute $\overline{c}^*(3) = 1$, as written in blue next to state 3. In step 2, we choose to remove edge $3 \to 6$ and recompute the optimal cost using the AIS without the removed edge. The resulting new optimal cost is 1, which is the same as the original optimal cost. Therefore, we know that the optimal action is the only remaining edge $3 \to 7$.

Once we find an optimal insertion action for every Z state using Algorithm 14, we construct optimal insertion strategy H in Algorithm 15 that contains all the actions at Y states and only the actions selected in the optimal strategy at Z states. The resulting H is a subgraph of the AIS and it will be used to build the optimal insertion automaton using Algorithm 12.

```
Algorithm 15: Find an optimal mean-cost insertion strategy
  input : AIS= $(Y \cup Z, E_o \cup 2^{E_i^*}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$ and weight function $w$
  output: Optimal strategy $H = (Y_H \cup Z_H, E_o \cup 2^{E_i^*}, f_H, y_{H,0})$
1 Let $y_{H,0} = y_0$. Set $Y_H = \{y_{H,0}\}$
2 for $u \in Y_H$ that has not been examined do
      for $e \in E_o$ do
          $f_H(u, e) := f_{AIS,yz}(u, e)$ if $f_{AIS,yz}(u, e)$ is defined
3 for $u \in Z_H$ that has not been examined do
      $f_H(u, e) := f_{AIS,yz}(u, e)$ where $e$ is the optimal action for $u$ computed using
      Algorithm 14
4 Go back to step 2 until all selected states have been examined
```

Theorem V.23. *Applying Algorithms 13, 14, 15 and 12 solves Problem V.12.*

Proof. Algorithms 13 and 14 follow the results in [66]. In Algorithm 15, all system actions are chosen and all insertion actions are optimized. Thus, the strategy is i-enforcing and optimal. Finally, the IA we obtain in Algorithm 12 is optimal because it is constructed from the strategy in Algorithm 15. □

In all, given the AIS= $(Y \cup Z, E_o \cup 2^{E_i^*}, f_{AIS,yz} \cup f_{AIS,zy}, y_0)$, computing the optimal mean-cost insertion function can be done in $O(n^4 |f_{AIS}| \log(\frac{|f_{AIS}|}{n}) W_{max})$, where n is the number of states in the AIS and $|f_{AIS}|$ is the number of transitions in $f_{AIS} = f_{AIS,yz} \cup f_{AIS,zy}$.

Example V.24. We have computed in Example V.22 the optimal action for state 3. In this example, we complete the optimal strategy using Algorithm 15 and build the optimal insertion automaton using Algorithm 12. In Algorithm 15, all insertion states other than state 3 in the weighted graph have degree 1. Thus, only edge $3 \rightarrow 6$ for state 3 needs to be removed. The resulting optimal state-based strategy H is the automaton without the shaded states in Figure 5.1(b). After H is obtained, we then follow Algorithm 12 to build the optimal IA from H, as it was done in Example V.18. The resulting optimal insertion automaton is shown in Figure 5.4.

Remark V.25. When finding the optimal action in Algorithm 14, there may be other actions that are as good as the selected one. As a consequence, there may be more than one solution to Problem V.12(b). Our algorithmic procedure returns one of them.

5.5 Conclusion

We have considered the problem of synthesizing an optimal insertion function for enforcing opacity. To quantify insertion functions, two quantitative objectives have been considered: the *maximum total cost* and the *maximum mean cost*. For each cost, we have developed an algorithm that computes the cost value of a given insertion function. We have also presented algorithms that synthesize an optimal insertion function, for each cost. Specifically, we first minimize the maximum total cost and determine if an optimal total-cost insertion function exists. If such an optimal one exists, we synthesize an optimal total-cost insertion function. Otherwise, we synthesize an optimal mean-cost insertion function. The synthesis algorithms presented in this chapter were developed by adapting and customizing results in game theory for minimax games and mean payoff games on weighted automata. Finally, we have encoded the resulting optimal insertion function as an I/O automaton.

CHAPTER VI

Case Study: Ensuring Privacy in Location-Based Services Using Opacity Techniques

The development of network and mobile devices has stimulated the rapid growth of networked services based on users' locations. Such services, called Location-Based Services (LBS), provide personalized and timely information to users by exploiting their real-time location information. Examples of LBS applications include searching for nearby restaurants, recommending in-store coupons for nearby shops, and providing turn-by-turn navigation instructions.

While LBS provides much convenience to users, they have also raised security and privacy concerns. The attacker, which is commonly assumed to reside on the LBS server, uses queries received from the users to infer sensitive information about the query senders [52]. Many prior studies have proposed to address LBS privacy by sending "cloaking queries" that contain coarser location information; see e.g., [25, 37, 21]. However, this method has been shown to be insufficient when users continuously make queries. The attacker, by tracking the user's continuous moving trajectory, can figure out the real user or the user's real location in the given cloaking query if other candidates have inconsistent trajectories [7]. Some works such as [63, 38] have addressed privacy in such dynamic settings. However, their approaches do not use formal methodology. Finally, we refer to [52] for a comprehensive overview of the recent schemes for protecting LBS users privacy.

In this chapter, we consider the problem of concealing the *current location* of the LBS user when the user continuously makes queries. We show that this problem can be formally addressed using opacity techniques in discrete event systems. Specifically, a nondeterministic finite-state automaton is used to capture the mobility patterns of the user. The attacker (i.e., the server) is assumed to know *a priori* the user's mobility patterns but only observes the location information in the received queries at real-time. We label the transitions of the automaton by the location information in the queries. To characterize location privacy, we adapt the notion of current-state opacity and introduce a related new notion called *current-state anonymity* that captures the observer's inability to know for sure the current state of the automaton. With the technique for verifying opacity, we show that sending cloaking queries to the server can still reveal the exact location of the user. To enforce location privacy, we synthesize an insertion function, using the synthesis algorithm developed in Chapter IV, that inserts fake queries into the cloaking query sequences. Such a insertion function inserts fake queries in a way that the resulting query sequences, as received by the server, are always consistent with the mobility model of the users and provably ensure privacy of the user's current location. Finally, to minimize the overhead from fake queries, we apply the optimization algorithm in Chapter V to design an optimal insertion function that introduces minimum average number of fake queries.

Using fake queries for location obfuscation has been proposed in the LBS privacy literature; see e.g., [29, 38]. The work in [29] generates random fake queries without considering the user's mobility patterns, and thus has raised the question of how convincing fake queries are. The algorithm in [38] relies on mobility patterns. However, it considers the privacy of the user instead of the privacy of the user's current location.

The remaining sections of this chapter are organized as follows. Section 6.1 introduces the common LBS architecture and the privacy concerns. Section 6.2 formalizes the notion of Current-State Anonymity in DES. In Section 6.3, we construct an automaton model from a set of mobility patterns on the Central Campus of the University of Michigan; this example is then used as a running example in the remaining two sections. We show in Section 6.4 how to verify location privacy using techniques for opacity verification. Then, in Section 6.5, we present the construction of an optimal insertion function for enforcement of location privacy. Finally, Section 6.6 concludes this chapter. Most of the results in this chapter also appear in [62]. We acknowledge Karthik Abinav Sankararaman for constructing the mobility model in this chapter and optimization the codes.

6.1 Location-Based Services

We consider the common LBS architecture described in [52]. The LBS architecture, as illustrated in Figure 6.1, consists of four major components: mobile devices, positioning systems, communication networks, and the LBS server. The user makes queries from his/her mobile devices. The location information in the queries is obtained via positioning systems such as the Global Positioning System (GPS). To protect privacy, the user identification in the queries is replaced with an untraceable pseudonym. The queries and their responses are transmitted between the user and the LBS server via communication networks.

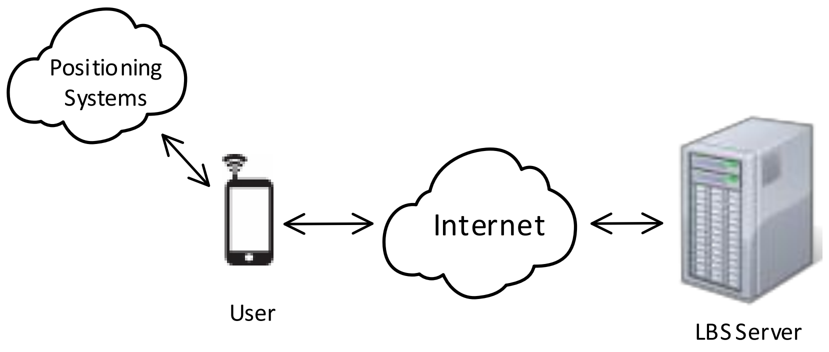


Figure 6.1: The common LBS architecture

6.1.1 Privacy Concerns and the Attack Model

We assume the attack model that is commonly used in the LBS community. The LBS server is a malicious observer and other components are benign. It may have such malicious intent for commercial purposes, or because they have been compromised. Specifically, we consider one attack scenario discussed in [53] where the server, i.e., the *attacker* in this chapter, knows the statistical information of users' moving patterns but is not aware of users' real-time location. The attacker relies on the location information it receives to perform attacks.

Two privacy notions for LBS have been defined, depending on what type of information is of concern: location privacy and query privacy. Location privacy is to prevent inferring the real locations where queries are made from the queries themselves. See, e.g., [29, 21, 15, 53]. Query privacy is to seclude the private attributes, such as the user's real identity, embedded in the queries. See, e.g., [25, 7, 37, 15, 63, 38]. In this work, we assume that the attacker aims to associate a user query with its real location and focus on location privacy only.

6.1.2 The Anonymizer Framework

To protect both query and location privacy, the pioneering work of [25] proposed to use *location anonymizers*. This technique has become the most popular technique for protecting privacy in LBS; see, e.g., [37, 21, 24]. A location anonymizer, as shown in Figure 6.2, generalizes the accurate location in a given user's original query to a *cloaking* region where $k-1$ other potential or active users reside. Implemented on a trusted third party, the anonymizer receives queries from users and forwards the cloaking queries to the LBS server.

This typical cloaking technique protects privacy for one-time queries only, but may fail in a dynamic environment where users continuously make queries [7]. In Section 6.4, we will show this result by using opacity techniques from DES. Then, we will show in Section 6.5 how to enforce location privacy using the opacity enforcement technique developed in Chapters IV and V.

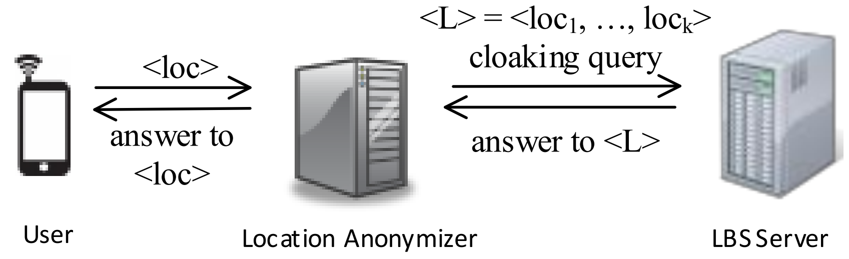


Figure 6.2: The traditional anonymizer framework

6.2 Current-State Anonymity in Discrete Event Systems

To fit the application of location privacy in LBS, we adapt the notion of current-state opacity (CSO) and propose a new notion called *Current-State Anonymity* (CSA). Specifically, CSO conceals the occurrence of the current state being at a given secret state, and CSA hides the occurrence of the current state being at any state. Let us first recall the definition of current-state opacity.

Definition VI.1 (Current-State Opacity)**.** Given system $G = (X, E, f, X_0)$, projection P, and the set of secret states $X_S \subseteq X$, current-state opacity holds if $\forall i \in X_0$ and $\forall t \in \mathcal{L}(G, i)$ such that $f(i, t) \subseteq X_S$, $\exists j \in X_0$, $\exists t' \in \mathcal{L}(G, j)$ such that: (i) $f(j, t') \cap (X \setminus X_S) \neq \emptyset$ and (ii) $P(t) = P(t')$.

We now adapt the notion of CSO and formally define current-state anonymity.

Definition VI.2 (Current-State Anonymity)**.** Given system $G = (X, E, f, X_0)$ and projection P, the system is current-state anonymous if $\forall i \in X_0$ and $\forall t \in \mathcal{L}(G, i)$ such that $f(i, t) = \{x\} \subseteq X$, $\exists j \in X_0$, $\exists t' \in \mathcal{L}(G, j)$, $\exists x' \neq x$ such that: (i) $x' \in f(j, t')$ and (ii) $P(t) = P(t')$.

Current-state anonymity is relevant to the widely-used notion of *k-anonymity* [55] when $k = 2$. It can be thought as CSO with multiple pairs of secret and non-secret states. Specifically, a given system is CSA if it is CSO with respect to $X_S = \{i\}, X_{NS} = X \setminus \{i\}, \forall i \in X$. The similarity between the two notions allows us to use the current-state estimator of G, which is also the observer automaton of G as defined in [10], to verify CSA. Hereafter, we denote the current-state estimator of G by $\mathcal{E}_G$. The following proposition then follows immediately.

Proposition VI.3. A given automaton G is current-state anonymous if and only if no state in $\mathcal{E}_G$ is a singleton.

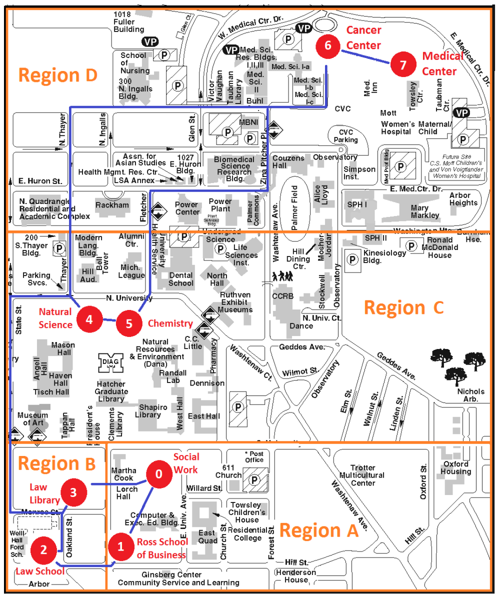
6.3 Automata Models for LBS

We consider the LBS framework where an anonymizer is used. The attacker has statistical information about users' mobility patterns, but can only perform attacks using the cloaking information sent to the server.

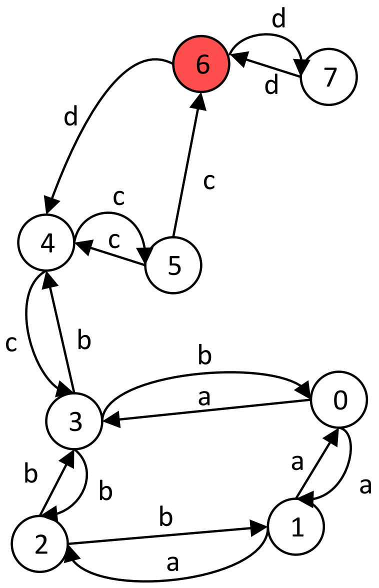
To map LBS privacy to opacity problems in DES, one key element is to build a finite-state automaton model that is consistent with the knowledge of the attacker (i.e., the server). We discretize the physical map and capture users' mobility patterns in a finite-state automaton. Specifically, we show our modeling methodology using a set of walking paths on the Central Campus of the University of Michigan. The constructed automaton will be our running example that illustrates the use of opacity techniques for location privacy. Shown in Figure 6.3(a) is the map of the Central Campus. We select eight locations as states, marked in red in the figure. Mobility patterns, as shown in blue lines, define transitions between states. Regions A, B, C, D are cloaking regions precomputed by the anonymizer.[1] We omit the unobservable details and label transitions by the cloaking information received by the server. Specifically, transitions are labeled by the cloaking regions of their source nodes, meaning that the user makes queries when s/he is about to move to the next location. The users

[1][53] has shown that computing cloaking regions in real-time does not improve privacy.

can start their walking paths from any location and thus $X_0 = X$. The constructed model is the nondeterministic finite-state automaton shown in Figure 6.3(b).



(a) The University of Michigan Central Campus map



(b) The constructed automaton model G

Figure 6.3: The campus map and the constructed automaton model

6.4 Verification of Location Privacy

To verify if the given moving patterns have location privacy, we need to know the server's location estimates and check if any estimate contains only one single location. Our methodology for constructing G from the moving patterns allows us to formulate location privacy as current-state anonymity. Location privacy holds if and only if the constructed G is current-state anonymous. As stated in Proposition VI.3, current-state anonymity can be verified using the current-state estimator $\mathcal{E}_G$. The moving patterns have location privacy if and only if no estimate state is a singleton.

Let us go back to the Central Campus example in Figure 6.3(a). We construct the current-state estimator $\mathcal{E}_G$ in Figure 6.4 to verify current-state anonymity. Before verifying, we first look at all estimate states in $\mathcal{E}_G$ that are reached by single events from the initial state of $\mathcal{E}_G$, such as state $\{4, 6, 7\}$ reached by event d. It can be found that no such state reveals the true location of the user. Because each cloaking region covers two distinct point locations and no such two locations lead to only one single location, location privacy for one-time queries can be guaranteed. However, to perform full verification, we need to examine the complete structure of $\mathcal{E}_G$. This corresponds to the server's knowledge in a dynamic environment where users continuously make queries. The estimate state $\{6\}$ shows that the user will reveal his/her true location at state 6 (i.e., Cancer Center) after querying sequences such as *cdd*. Hence, current-sate anonymity does not hold. This revelation is because the server knows that consecutive queries d's can only be made between states 6 and 7 and that the user came from region C from the first received query. This also shows that the anonymizer's cloaking technique does not necessarily provide location privacy.

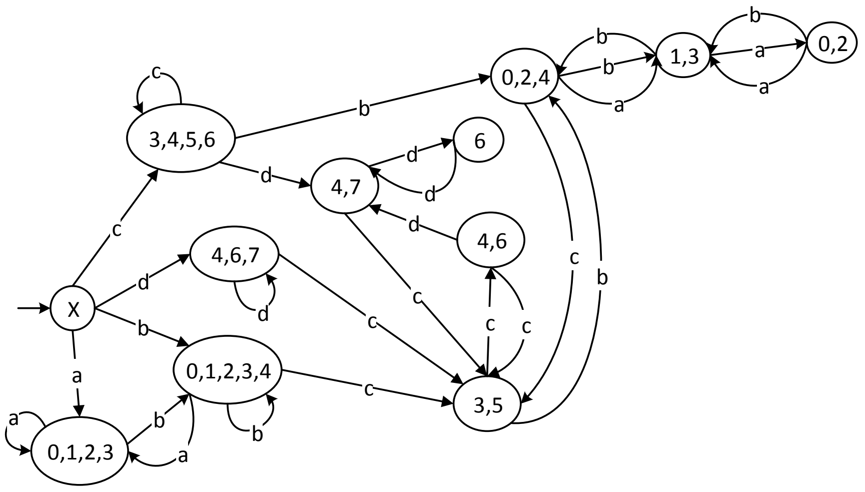


Figure 6.4: The current-state estimator $\mathcal{E}_G$ of G

6.5 Enforcement of Location Privacy

6.5.1 I-Enforcing Insertion Functions

To resolve the location revelation identified in Section 6.4, we propose to add to the anonymizer an insertion function that inserts fake queries to the user's original query sequence. An insertion function, as formally defined in Chapter IV for opacity enforcement, is placed between the system and the outside attacker that modifies the system's output behavior by inserting fictitious events. In the framework of LBS, we place the insertion function at the output of the anonymizer, as shown in Figure 6.5. It inserts fake queries to the cloaking queries and drops their replies without affecting the quality of the server's replies to real queries.

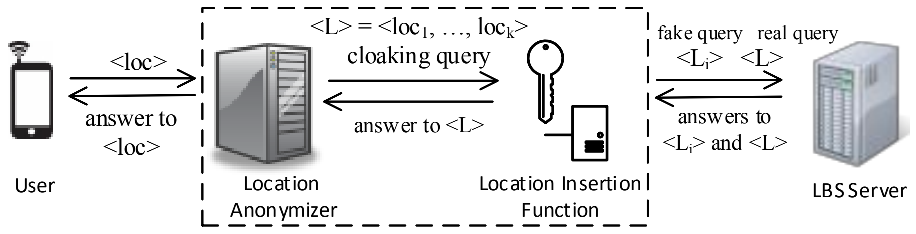
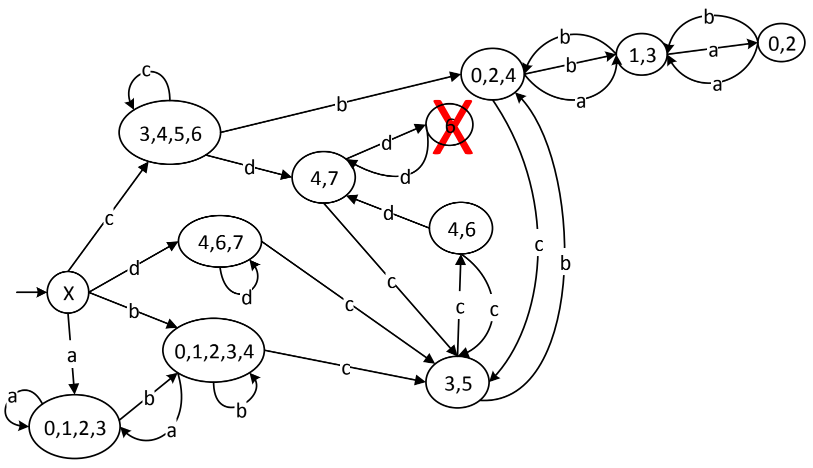


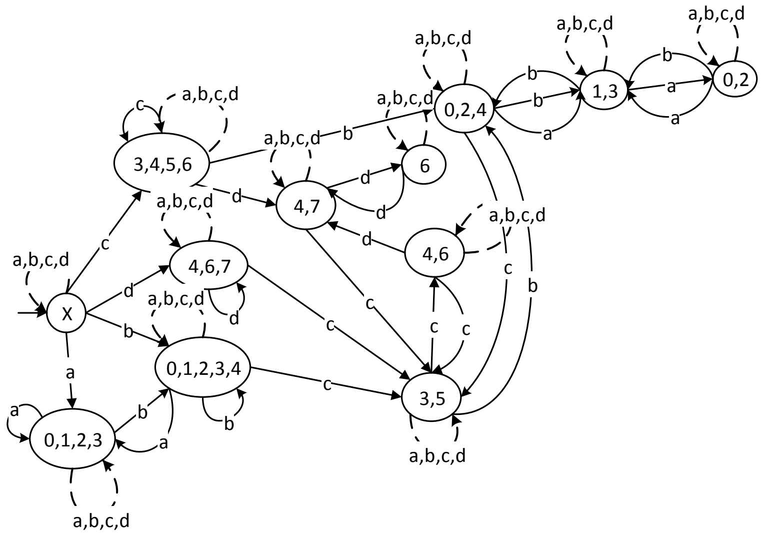
Figure 6.5: The proposed location insertion mechanism

Recall from Chapter IV that to enforce opacity, insertion functions need to satisfy a property called *i-enforceability.* An insertion function f_I is *i-enforcing* if it is (i) safe: every output behavior from f_I is in the safe language L_{safe}, and (ii) admissible: f_I does not block or change any output from the system. Here, we slightly adapt the i-enforceability property and redefine *the i-enforceability property for enforcing CSA*. Specifically, we redefine L_{safe} to be all observable strings that do not reveal the current state of the system. That is, it contains all strings in $\mathcal{L}(\mathcal{E}_G)$ that do not visit singleton states (i.e., state $\{6\}$ for $\mathcal{E}_G$ in Figure 6.4). Because the synthesis of an i-enforcing insertion function is based on L_{safe}, once L_{safe} for enforcing CSA is defined, we can follow the procedure in Chapter IV to synthesize an insertion function that enforces current-state anonymity. Consequently, the insertion function f_I synthesized from the AIS will be i-enforcing with respect to CSA. That is, (i) every output behavior from f_I does not lead to a singleton state in $\mathcal{E}_G$ and (ii) f_I does not block or change any query from the anonymizer.

To synthesize an insertion function that enforces CSA, we construct the AIS of G by following the procedure in [58], adapted for our L_{safe}. We begin with constructing the i-verifier V that recognize all the safe insertions. For the Central Campus example, we build the desired estimator $\mathcal{E}^d$ in Figure 6.6(a), which removes state $\{6\}$ from $\mathcal{E}_G$, and the feasible estimator $\mathcal{E}^f$, which includes all possible insertions, and synchronize them. After following the complete procedure in Chapter IV, we build the AIS



(a) The desired estimator $\mathcal{E}^d$



(b) The feasible estimator $\mathcal{E}^f$

Figure 6.6: The desired estimator and the feasible estimator for G in Figure 6.3(b)

enumerates all i-enforcing insertion functions using a game structure that describes the interaction of the system (i.e., the user's continuous queries from the anonymizer) and the insertion function. The entire AIS for the Central Campus example has 84 states. We show in Figure 6.7 the AIS drawn using DESUMA [17]. For illustration

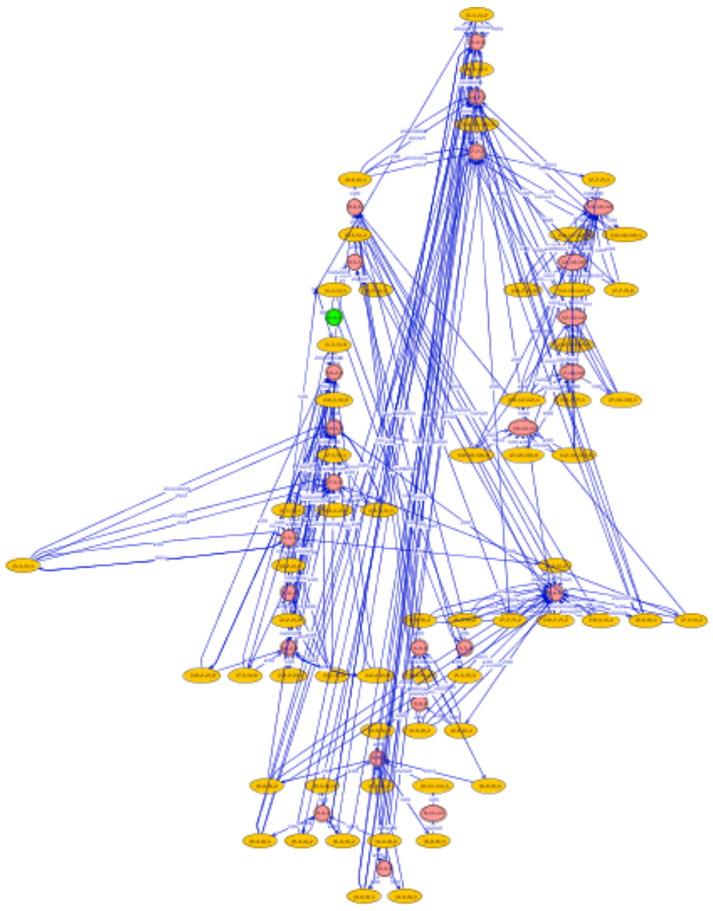


Figure 6.7: The AIS of the Central Campus example, drawn using DESUMA

purpose, let us look at a partial structure of the abstracted AIS that is shown in Figure 6.8. As can be seen in the figure, the AIS is a bipartite graph with "square" states and "round" states. The shapes of these states tell us whether the anonymizer or the insertion function is acting in the game. At square states, the anonymizer enumerates all possible user queries according to the moving patterns. For example, initially at state 0, the anonymizer can output queries a, b, c, d. At round states, the insertion function enumerates all i-enforcing insertion choices, determined from the construction procedure of the AIS, that respond to the queries from the anonymizer. For example, after the anonymizer outputs d, at state 4, the insertion function can insert $\varepsilon, b_i, c_i, b_ic_ic_i$. By listing all actions of the two players in this manner, the AIS enumerates all i-enforcing insertion functions. To synthesize one i-enforcing insertion function, one needs to select all edges from the square states (in order to consider all user's queries) and one insertion edge for each round state that is reached. In the next section, we will discuss how to select insertion edges and synthesize one *optimal* insertion function from the AIS.

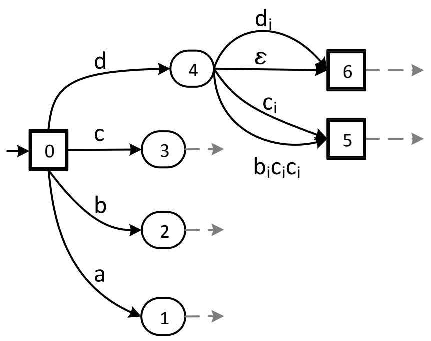


Figure 6.8: Partial representation of the abstracted AIS (7 out of 84 states)

6.5.2 Optimal Insertion Functions

One can synthesize an i-enforcing insertion function by "randomly" selecting insertion edges at the round states in the AIS. But as inserting fictitious queries introduces extra delay and consumes energy, a more interesting problem is to minimize the overhead cost introduced by insertion, i.e., to synthesize an *optimal* insertion function.

In the Central Campus example, we assign the same unit cost to each inserted query, for the sake of simplicity. We then employ the optimization algorithms developed in Chapter V to synthesize an optimal insertion function from the AIS. Specifically, we compute $V_l(0) = 28220$ using Equation (5.2), where $l = n^2 W_{max} = 84^2 \cdot 4 = 28224$. Because $V_l(0) \geq nW_{max} = 336$, we conclude from Theorem V.17 that there is no optimal total-cost insertion function. Hence, we solve for an optimal mean-cost insertion function by applying Algorithms 13, 14, 15 and 12 in Chapter V. The optimal maximum mean cost is found to be 2 and the resulting insertion automaton is shown in Figure 6.9. The insertion function encoded in Figure 6.9 will provably guarantee that visiting location 6 is never revealed.

Let us look at how this insertion automaton modifies the problematic query sequence cdd. The insertion automaton modifies cdd to cdc_ic_id by inserting c_ic_i, which induces the intruder to generate estimate $\{4, 7\}$. It is worth noticing that, for this particular query sequence, the intruder's new inference does not even include the true actual location 6. However, this is not generally true.

We first observe that no i-enforcing insertion function has a *finite* worst-case total cost for the case of G of Figure 4.4(a). That is, to enforce location privacy in this example, an insertion function needs to continuously add fictitious queries as the user makes queries (note that there are many cyclic paths that return to singleton state $\{6\}$ in Figure 6.4.)

Hence, we consider the quantitative objective for optimization purposes to be the long-run average insertion cost (i.e., per user query), in the worst case. (See [60] for further technical details about this cost structure.) The minimum value that we find in our example is 2, meaning that at most two fake queries per user query need to be inserted. Using this value, we then synthesize from the AIS an insertion function f_I that achieves this optimal value. The result for our example is shown in Figure 6.9, where f_I is encoded as an I/O automaton where the input labels are queries from the anonymizer and the output labels are the modified queries with insertion. Events with subscript i denote fake queries from the insertion function.

Let us now look at query sequence cdd, which is found in Section 6.4 to reveal the true location of the user. We can see from Figure 6.9 that the optimal insertion function will modify cdd to cdc_ic_id. The server does not distinguish fictitious and genuine queries; i.e., c and c_i are indistinguishable by the LBS. Thus, when the server receives cdc_ic_id, it interprets the sequence as $cdccd$ and infers that the user is at $\{4, 7\}$, while the true location is 6. Hence, location privacy is enforced.

Remark VI.4. While we have synthesized an insertion function in the anonymizer framework, the insertion mechanism does not require an anonymizer. To develop

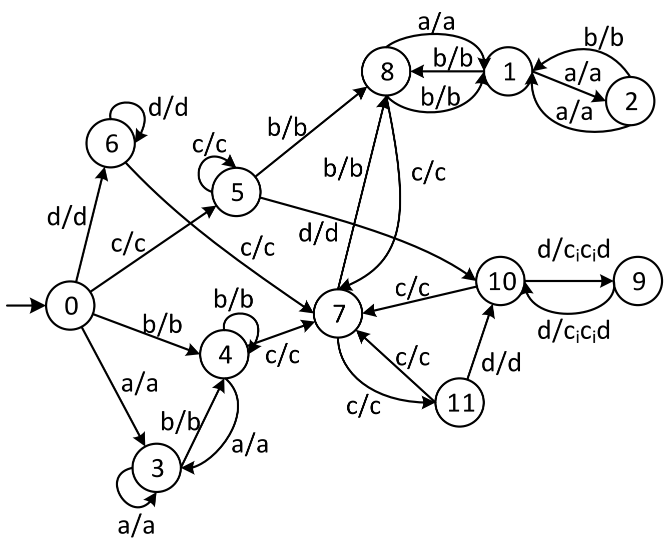


Figure 6.9: The I/O automaton representation of the optimal insertion function

the insertion mechanism without an anonymizer, one needs to modify the automaton model by labeling transitions using the original queries instead of the cloaking queries. An i-enforcing insertion function can then be constructed by following the same procedure. Note that each query is made when the user is about to move and is labeled by the region of the source location.

6.6 Conclusion

We have considered location privacy in Location-Based Services in the context of opacity problems in DES and presented a formal approach to solve location privacy problems. To characterize location privacy, we have adapted the notion of current-state opacity and defined *current-state anonymity.* We have developed a modeling methodology that capture users' mobility patterns using an automaton model. By using the current-state estimator to verify current-state anonymity, we have illustrated that the traditional anonymizer framework is in general insufficient to protect location privacy in a dynamic environment. To enforce location privacy, we have proposed to use the mechanism of *insertion functions* developed in Chapter IV, that inserts fictitious queries in the cloaking query sequences. We have also shown how to synthesize an *optimal* insertion function to minimize the overhead costs caused by insertion.

CHAPTER VII

Conclusion and Future Work

7.1 Summary of Main Contributions

In this dissertation, we have studied opacity notions in Discrete Event Systems modeled as partially-observable and/or nondeterministic finite-state automata. In opacity problems, the system has a *secret*. The intruder is a passive observer of the system that wants to know whether or not the *secret* of the system has occurred. The system is opaque if "whenever the secret has occurred, there exists another *non-secret* behavior that is observationally equivalent."

We have focused on the verification and the enforcement of various notions opacity: current-state opacity, initial-state opacity, language-based opacity, and initial-and-final-state opacity. For each state-based notion of opacity, we have developed verification algorithms when there is one single intruder and when there are multiple intruders collaborating through a coordinator. When the given opacity notion fails to hold, we have proposed a novel enforcement mechanism based on the use of *i-enforcing insertion functions.* An insertion function, placed at the output of the system, inserts fictitious observable events to the system's output. The property of i-enforceability guarantees that the given insertion function enforces opacity without modifying the internal behavior of the system. We have developed a formal procedure that synthesizes an i-enforcing insertion function. Specifically, we have built a finite structure called the AIS that enumerates *all* i-enforcing insertion functions. The AIS provides a necessary and sufficient condition for the existence of an i-enforcing insertion function. When an i-enforcing insertion function exists for the given notion of opacity, we have developed an algorithmic procedure that synthesizes one insertion function.

Inserted events introduce overhead delays and consume bandwidth. Hence, we have assigned costs to inserted events and investigated the synthesis of an *optimal* insertion function with respect to the given cost criterion. We have solved two optimization problems, one with respect to the *maximum total cost* and the other with respect to the *maximum mean cost*, by exploiting the structure of the AIS. Specifically, we first minimize the maximum total cost and determine if an optimal total-cost insertion function exists. If such an optimal one exists, we synthesize an optimal total-cost insertion function. Otherwise, we synthesize an optimal mean-cost insertion function. Synthesis algorithms for either case have been developed by adapting results in game theory for minimax games and mean payoff games on weighted automata. The resulting insertion function is encoded as an I/O automaton. Finally, we have adapted and applied the above analysis and enforcement procedure to the problem of enforcing location privacy in location-based services.

7.2 Future Work

The research in this dissertation opens several research directions for future work. First, the synthesis of i-enforcing insertion functions in Chapter IV assumes that the intruder does not know the insertion functions at the outset. The i-enforceability property assures that the intruder would never figure out the existence of an insertion function. However, an interesting question that arises is what happens if the intruder knows the insertion function. We have discussed in Section 4.10 how the intruder's knowledge of insertion functions can affect the construction of an insertion function.

It would be interesting to adapt the synthesis procedure for i-enforcing insertion functions and design insertion functions that *enforce opacity regardless of whether the intruder knows the insertion function.*

Another possible extension is to study opacity notions in probabilistic settings. In the logical opacity problems, we do not use a probabilistic model for the system dynamics; opacity holds when plausible deniability of the secret is guaranteed. However, in this context, the intruder's confidence of whether the secret has occurred or not is not characterized. It would be of interest to extend the investigation of opacity to the notion of "stochastic opacity", which captures the scenario where the intruder has prior knowledge of the system's transition probabilities and infers that the occurring behavior is the *most probable* one based on its observation. Stochastic opacity would hold if *given any observable behavior from the system, there is a non-secret behavior that is more probable.* This notion is similar, but different from the opacity notions in probabilistic settings defined in [6, 47]. Algorithms for verifying probabilistic opacity notions need to developed. When probabilistic opacity fails to hold, we could consider the enforcement of probabilistic opacity by using a *stochastic insertion function*, which inserts events according to some specifications but randomizes the insertion when there are multiple choices. The use of stochastic insertion functions would allow to enforce a larger class of stochastic opacity problems.

It would be also interesting to study opacity notions in modular settings. In such settings, the system G is the composition of N subsystems; i.e., $G = G_1||G_2||\ldots||G_N$. Each subsystem G_i has its own secret. While each G_i may be individually opaque, composing them can result in G that reveals the local secret because some nonsecret strings are disabled in the composition. The authors of [44] have developed algorithms for verifying opacity in a modular manner. However, modular control algorithms that enforce opacity in system G have not been explored. We could develop local insertion strategies such that opacity of the whole system G is enforced. Such work would provide principles that facilitate the modular development of secure systems and ensure that adding secure subsystems into a secure system results in a secure system.

Last, more case studies are worth investigating. It would be of interest to apply the technique developed in this thesis to preserve privacy properties in other network security applications.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] R. Alur, P. Černỳ, and S. Zdancewic. Preserving secrecy under refinement. *Automata, Languages and Programming*, pages 107–118, 2006.

[2] F. Baccelli, G. Cohen, G. J. Olsder, and J. P. Quadrat. *Synchronization and linearity*, volume 3. Wiley New York, 1992.

[3] E. Badouel, M. Bednarczyk, A. Borzyszkowski, B. Caillaud, and P. Darondeau. Concurrent secrets. *Discrete Event Dynamic Systems: Theory and Aplications*, 17(4):425–446, 2007.

[4] M. Ben-Kalefa and F. Lin. Opaque superlanguages and sublanguages in discrete event systems. In *Proc. of the 48th IEEE Conference on Decision and Control*, pages 199–204. IEEE, 2009.

[5] M. Ben-Kalefa and F. Lin. Supervisory control for opacity of discrete event systems. In *Proc. of the 49th Annual Allerton Conference on Communication, Control, and Computing*, pages 1113–1119. IEEE, 2011.

[6] B. Bérard, J. Mullins, and M. Sassolas. Quantifying opacity. In *Proc. of the 7th International Conference on the Quantitative Evaluation of Systems*, pages 263–272. IEEE, 2010.

[7] C. Bettini, X. S. Wang, and S. Jajodia. Protecting privacy against location-based personal identification. In *Secure Data Management*, pages 185–199. Springer, 2005.

[8] J. Bryans, M. Koutny, and P. Y. A. Ryan. Modeling opacity using Petri nets. *Electronic Notes in Theoretical Computer Science*, 121:101–115, 2005.

[9] J. W. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan. Opacity generalized to transition systems. *International Journal of Information Security*, 7(6):421–435, 2008.

[10] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems, 2nd Edition.* Springer, 2008.

[11] F. Cassez, J. Dubreil, and H. Marchand. Synthesis of opaque systems with static and dynamic masks. *Formal Methods in System Design*, pages 1–28, 2012.

[12] F. Cassez and S. Tripakis. Fault diagnosis with static and dynamic observers. *Fundamenta Informaticae*, 88(4):497–540, 2008.

[13] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[14] S. Chédor, C. Morvan, S. Pinchinat, H. Marchand, et al. Analysis of partially observed recursive tile systems. In *11th Int. Workshop on Discrete Event Systems*, pages 265–271, 2012.

[15] C.-Y. Chow and M. F. Mokbel. Enabling private continuous queries for revealed user locations. In *Advances in Spatial and Temporal Databases*, pages 258–275. Springer, 2007.

[16] R. Debouk, S. Lafortune, and D. Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete event systems. *Discrete Event Dynamic Systems: Theory and Aplications*, 10(1-2):33–86, 2000.

[17] DESUMA Team. DESUMA software tool, 2014. `http://www.eecs.umich.edu/umdes/toolboxes.html`.

[18] J. Dubreil, P. Darondeau, and H. Marchand. Opacity Enforcing Control Synthesis. In *Proc. of the 9th International Workshop on Discrete Event Systems*, page 2835, Goteborg, Sweden, May 2008. IEEE.

[19] J. Dubreil, P. Darondeau, and H. Marchand. Supervisory control for opacity. *IEEE Transactions on Automatic Control*, 55(5):1089–1100, 2010.

[20] J. Dubreil, T. Jéron, and H. Marchand. Monitoring confidentiality by diagnosis techniques. In *Proc. of the 10th European Control Conference*, Budapest, Hungary, August 2009.

[21] M. Duckham and L. Kulik. A formal model of obfuscation and negotiation for location privacy. In *Pervasive Computing*, pages 152–170. Springer, 2005.

[22] A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8(2):109–113, 1979.

[23] Y. Falcone and H. Marchand. Runtime enforcement of K-step opacity. In *Proc. of the 52nd IEEE Conference on Decision and Control*, 2013.

[24] B. Gedik and L. Liu. Protecting location privacy with personalized k-anonymity: Architecture and algorithms. *IEEE Transactions on Mobile Computing*, 7(1):1–18, 2008.

[25] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proc. of the 1st International Conference on Mobile Systems, Applications and Services*, pages 31–42, 2003.

[26] N.B. Hadj-Alouane, S. Lafrance, F. Lin, J. Mullins, and M. Yeddes. On the verification of intransitive noninterference in mulitlevel security. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 35(5):948–958, 2005.

[27] T. Handel and M. Sandford. Hiding data in the OSI network model. In *Information Hiding*, pages 23–38. Springer, 1996.

[28] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete mathematics*, 23(3):309–311, 1978.

[29] H. Kido, Y. Yanagisawa, and T. Satoh. An anonymous communication technique using dummies for location-based services. In *Proc. of International Conference on Pervasive Services*, pages 88–97, 2005.

[30] K. Kobayashi and K. Hiraishi. Verification of opacity and diagnosability for pushdown systems. *Journal of Applied Mathematics*, 2013, 2013.

[31] R. Kumar and V.K. Garg. *Modeling and control of logical discrete event systems.* Kluwer Academic Publishers, 1995.

[32] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[33] F. Lin. Opacity of discrete event systems and its applications. *Automatica*, 47(3):496–503, 2011.

[34] F. Lin and WM Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, 1988.

[35] L. Mazaré. Using unification for opacity properties. *Proc. of the 4th IFIP WG1*, 7:165–176, 2003.

[36] D. McMorrow. Science of cyber-security. Technical report, DTIC Document, 2010.

[37] M. F. Mokbel, C.-Y. Chow, and W. G. Aref. The new Casper: Query processing for location services without compromising privacy. In *Proc. of the 32nd International Conference on Very Large Data Bases*, pages 763–774, 2006.

[38] A. Pingley, N. Zhang, X. Fu, H.-A. Choi, S. Subramaniam, and W. Zhao. Protection of query privacy for continuous location based services. In *Proc. of 2011 IEEE INFOCOM*, pages 1710–1718, 2011.

[39] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1), 1987.

[40] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):66–92, 1998.

[41] A. Saboori. *Verification and enforcement of state-based notions of opacity in discrete event systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2010.

[42] A. Saboori and C. N. Hadjicostis. Notions of security and opacity in discrete event systems. *Proc. of the 46th IEEE conference on Decision and Control*, pages 5056–5061, Dec 2007.

[43] A. Saboori and C. N. Hadjicostis. Verification of initial-state opacity in security applications of DES. *Proc. of the 9th International Workshop on Discrete Event Systems*, pages 328–333, May 2008.

[44] A. Saboori and C. N. Hadjicostis. Reduced-complexity verification for initial-state opacity in modular discrete event systems. In *Proc. of the 12th International Workshop of Discrete Event Systems*, volume 10, pages 78–83, 2010.

[45] A. Saboori and C. N. Hadjicostis. Verification of k-step opacity and analysis of its complexity. *IEEE Transactions on Automation Science and Engineering*, 8(3):549–559, 2011.

[46] A. Saboori and C. N. Hadjicostis. Opacity-enforcing supervisory strategies via state estimator constructions. *IEEE Transactions on Automatic Control*, 57(5):1155–1165, 2012.

[47] A. Saboori and C. N. Hadjicostis. Current-state opacity formulations in probabilistic finite automata. *IEEE Transactions on automatic control*, 59(1):120–133, Jan 2014.

[48] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *Automatic Control, IEEE Transactions on*, 40(9):1555–1575, 1995.

[49] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[50] F. B. Schneider. Blueprint for a science of cybersecurity. 2011.

[51] S. Schneider and A. Sidiropoulos. CSP and anonymity. In *Computer Security-ESORICS 96*, pages 198–218. Springer, 1996.

[52] K. G. Shin, X. Ju, Z. Chen, and X. Hu. Privacy protection for users of location-based services. *Wireless Communications, IEEE*, 19(1):30–39, 2012.

[53] R. Shokri, C. Troncoso, C. Diaz, J. Freudiger, and J.-P. Hubaux. Unraveling an old cloak: k-anonymity for location privacy. In *Proc. of the 9th ACM Workshop on Privacy in the Electronic Society*, pages 115–118, 2010.

[54] S. Shu, F. Lin, and H. Ying. Detectability of discrete event systems. *IEEE Transactions on Automatic Control*, 52(12):2356–2359, 2007.

[55] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[56] S. Takai and Y. Oka. A formula for the supremal controllable and opaque sublanguage arising in supervisory control. *SICE Journal of Control, Measurement, and System Integration*, 1:307–311, 2011.

[57] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[58] Y.-C. Wu and S. Lafortune. Enforcement of opacity properties using insertion functions. In *Proc. of the 51th IEEE Conference on Decision and Control*, pages 6722–6728, December 2012.

[59] Y.-C. Wu and S. Lafortune. Comparative analysis of related notions of opacity in centralized and coordinated architectures. *Discrete Event Dynamic Systems: Theory and Applications*, 23(3):307–339, September 2013.

[60] Y.-C. Wu and S. Lafortune. Synthesis of insertion functions for enforcement of opacity security properties. *Automatica*, 50(5):1336–1348, 2014.

[61] Y.-C. Wu and S. Lafortune. Synthesis of optimal insertion functions for opacity enforcement. Technical report, Submitted for publication, July 2014.

[62] Y.-C. Wu, K. A. Sankararaman, and S. Lafortune. Ensuring privacy in location-based services: An approach based on opacity enforcement. *Proc. of the 14th International Workshop of Discrete Event Systems*, pages 33–38, 2014.

[63] T. Xu and Y. Cai. Exploring historical location data for anonymity preservation in location-based services. In *Proc. of the IEEE INFOCOM*, pages 547–555, 2008.

[64] T.-S. Yoo and S. Lafortune. Polynomial-time verification of diagnosability of partially observed discrete-event systems. *IEEE Transactions on automatic control*, 47(9):1491–1495, 2002.

[65] B. Zhang, S. Shu, and F. Lin. Polynomial algorithms to check opacity in discrete event systems. In *Proc. of the 24th Chinese Control and Decision Conference*, pages 763–769. IEEE, 2012.

[66] U. Zwick and M. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1):343–359, 1996.