UNICA

UNIVERSITÀ
DEGLI STUDI
DI CAGLIARI

Ph.D. DEGREE IN

Electronic and Computer Engineering

Cycle XXXVII

TITLE OF THE Ph.D. THESIS

Opacity Verification and Fault Diagnosis in the Framework of Switching Output Automata

Scientific Disciplinary Sector(s)

ING-INF/04 - Automatica

Ph.D. Student:	Tianyu Liu
Supervisor	Prof. Alessandro Giua
Co-Supervisor	Prof. Carla Seatzu

Final exam. Academic Year 2023/2024
Thesis defence session: February 2026

Abstract

Security and reliability in cyber-physical systems depend not only on *which* operational mode the system occupies, but also on *how long* it remains there. In practice, external observers see piecewise-constant output signals (power consumption levels, alarm indicators, aggregate measurements) together with their durations, rather than internal state transitions or continuous trajectories. Classical discrete-event models capture event sequences but lack native timing semantics, while dense-time formalisms such as timed automata face undecidability or prohibitive complexity for verification tasks involving both timing constraints and partial observability.

This dissertation introduces the Switching Output Automaton (SOA) framework to address this gap. SOA models what observers see in practice: output symbols and their durations. The key innovation is a minimal dwell time constraint that requires any two consecutive events to be separated by at least a fixed positive duration. This constraint reflects physical reality and enables decidable verification by bounding the rate of observable changes. SOA combines continuous-time semantics with a set-valued output function that creates observational ambiguity independent of timing, allowing multiple states to share output symbols.

We apply this framework to two timing-aware security and reliability problems. First, for *opacity verification*, we establish decidable procedures for both current-state opacity (where secrets are discrete states) and timed opacity (where secrets are global-state-duration triples specifying which combinations of discrete state, output, and dwell time must remain confidential). For timed opacity, verification proceeds by constructing a secret-dependent evolution automaton that applies fine-grained time discretization only to vulnerable global states, then building an observer automaton through subset construction, and finally reducing opacity to reachability analysis. This approach achieves decidability without the severe restrictions required for timed automata, keeping complexity within the observer-based determinization profile familiar from untimed discrete-event systems.

Second, for *timed fault diagnosis*, we extend SOA to a Switching Output Automaton with Faults (SOAF) that models faults enabled only during specific dwell-time windows in fault-prone states. This captures degradation phenomena where failures emerge from sustained exposure to stress conditions rather than instantaneous events. Diagnosis proceeds through a four-stage construction: logical state discretization preserving fault-window boundaries, evolution automaton with faults construction tracking nominal and fault transitions, fault recognizer synthesis with observation mappings, and subset determinization yielding an online diagnoser that classifies observation sequences as normal, faulty, or uncertain with sound and complete detection guarantees.

Three case studies validate the framework across safety-critical domains. A smart water supply system verifies current-state opacity under aggregate power monitoring, demonstrating that critical pump-valve configurations remain hidden from adversaries observing power consumption. A patient monitoring system verifies timed opacity for healthcare privacy, ensuring prolonged critical-condition patterns remain indistinguishable from benign fluctuations through network observations. A battery management system applies timed fault diagnosis to thermal runaway detection, illustrating how health-dependent fault window parameterization reduces detection delay. To support reproducibility, we provide an open-source implementation of the verification framework.

Contents

List of Figures

List of Tables

List of Acronyms

2-EXPTIME	Double-Exponential Time
ADC	Analog-to-Digital Converter
BFS	Breadth-First Search
BMS	Battery Management System
CPS	Cyber-Physical Systems
CTLA	Constant-Time Labeled Automaton
CTMC	Continuous-Time Markov Chain
DAC	Digital-to-Analog Converter
DBM	Difference Bound Matrix
DES	Discrete-Event Systems
DFA	Deterministic Finite Automaton
DTS200	Amira DTS200 Smart Water Supply Testbed
EA	Evolution Automaton
EAF	Evolution Automaton with Faults
ECM	Equivalent Circuit Model
EKF	Extended Kalman Filter
EXPSPACE	Exponential Space
EXPTIME	Exponential Time
FEUP	Faculdade de Engenharia da Universidade do Porto
GDPR	General Data Protection Regulation
ICU	Intensive Care Unit
IDES	Integrated Discrete Event Systems Tool

IRTA	Integer-Reset Timed Automata
MDP	Markov Decision Process
MILP	Mixed Integer Linear Programming
MTTF	Mean Time To Failure
NFA	Nondeterministic Finite Automaton
NILM	Non-Intrusive Load Monitoring
NIST	National Institute of Standards and Technology
PLC	Programmable Logic Controller
PSPACE	Polynomial Space
PVDF	Polyvinylidene Fluoride
RTA	Real-Time Automata
SCADA	Supervisory Control and Data Acquisition
SEI	Solid Electrolyte Interphase
SMT	Satisfiability Modulo Theories
SOA	Switching Output Automaton
SOAF	Switching Output Automaton with Faults
SOC	State of Charge
SOH	State of Health
SWSS	Smart Water Supply System
TA	Timed Automata
TIDES	Time-Interval Discrete Event Systems
TPN	Time Petri Net
UKF	Unscented Kalman Filter
UPPAAL	Uppsala-Aalborg Model Checker
ZOH	Zero-Order Hold

List of Symbols

General Notation

Notation	Description
$\mathbb{N}$	Set of natural numbers (including 0)
$\mathbb{N}_+$	Set of positive natural numbers
$\mathbb{R}$	Set of real numbers
$\mathbb{R}_{\geq 0}$	Set of non-negative real numbers
$\mathbb{R}_{>0}$	Set of positive real numbers
2^S	Power set of set S
$\lvert S \rvert$	Cardinality of set S
ε	Empty string or silent/unobservable transition
δ	Minimal dwell time (positive real constant)
$\lfloor \cdot \rfloor$	Floor function

Automata Components

Switching Output Automaton (SOA)

Notation	Description
G	Switching Output Automaton
X	Finite set of discrete states
Y	Output alphabet (set of output symbols)
B	Set of arcs (transitions between states)
$h : X \to 2^Y$	Output function (maps states to sets of outputs)
x_0	Initial state
y_0	Initial output symbol
Q	Set of global states (pairs (x, y))
(x, y)	Global state (discrete state x with output y)
$\sigma(x)$	Set of direct successors of state x

Evolution Automaton

Notation	Description
G_e	Evolution Automaton
X'	Set of waiting discrete states
X''	Set of ready discrete states
X_e	Extended discrete state space ($X' \cup X''$)
$\bar{Q}$ or Q_e	Set of logical states
$(x,y)_j$	Logical state (global state (x,y) with dwell-time index j)
Y_e	Alphabet of evolution automaton ($Y \cup \{\delta\}$)
Δ	Transition relation
$\bar{q}_0$ or $q_{e,0}$	Initial logical state
$\mathscr{R}(q)$	Maximum interval index for global state q

Observer

Notation	Description
G_{obs} or $\text{Obs}(G)$	Observer automaton
Z	Set of observer states (belief states)
δ_o or Δ_o	Observer transition function
z_0	Initial observer state
$D_\varepsilon(q)$	ε-closure of state q

Opacity Verification

Notation	Description
S	Timed secret (set of secret configurations)
X_s	Set of secret states
Q_v	Set of vulnerable global states
$\mathscr{I}_v(q)$	Secret dwell-time interval for global state q
(q,t)	Secret configuration (global state q at dwell time t)
$\mathscr{L}_{\text{secret}}(q)$	Set of secret logical states for global state q
$\mathscr{R}'(q), \mathscr{R}''(q)$	Interval bounds for secret partitioning

Fault Diagnosis

SOAF (Switching Output Automaton with Faults)

Notation	Description
G_f	Switching Output Automaton with Faults (SOAF)
B_f	Set of fault arcs
X_f	Set of fault-prone states
Q_f	Set of fault-prone global states
$\mathscr{I}(q,b_f)$	Fault occurrence intervals for global state q and fault arc b_f
$\sigma_f(x)$	Set of successors reachable via fault transitions from x
$E(q,j,b_f)$	Fault enablement function (determines when fault arc is active)

Evolution Automaton with Faults (EAF)

Notation	Description
G_{ef}	Evolution Automaton with Faults
Σ or $\{n, f\}$	Event alphabet (normal, fault)

Fault Monitor, Recognizer and Diagnoser

Notation	Description
M	Fault monitor automaton
X_M	Fault monitor state set ($\{N, F\}$)
$\mathrm{Rec}(G_{ef})$	Fault recognizer
X_R	Set of recognizer states
Δ_R	Recognizer transition relation
$\mathscr{O}$	Observation alphabet ($Y \cup \{\delta\}$)
$\mathscr{P}$	Observation mapping function ($\Delta_R \to \mathscr{O} \cup \{\varepsilon\}$)
γ	Fault label (N for normal, F for faulty)
G_{diag}	Diagnoser automaton
$\varphi(z)$	Diagnostic evaluation function (N, F, or U for uncertain)

Runs and Observations

Notation	Description
t_i	Time instant of i-th state transition
$\tau_j^{(i)}$	Time instant of j-th output switch in state $x^{(i)}$
$\mathscr{T}$	Event time set ($\{t_i\} \cup \{\tau_j^{(i)}\}$)
$x^{(i)}$	Discrete state during i-th phase
$y_j^{(i)}$	Output value during j-th period in state $x^{(i)}$
m_i	Number of output switches in state $x^{(i)}$
$y_{\mathrm{obs}}(t)$	Output observation signal (maps time to output)
(y, τ)	Timed output symbol (output y for duration τ)
ω	Output behavior (sequence of timed output symbols)
$L(G)$ or $L_\delta(G)$	Language of automaton G (set of valid behaviors)
u	Logical observation sequence
ψ	Logical observation mapping

Complexity Classes

Notation	Description
PSPACE	Polynomial space complexity class
EXPTIME	Exponential time complexity class
EXPSPACE	Exponential space complexity class
2-EXPTIME	Double-exponential time complexity class

Battery Management System (Case Study)

Notation	Description
G_{BMS}	Battery Management System SOAF model
I	Discharge current (amperes)
T	Battery cell temperature
$\dot{Q}_{\text{gen}}$	Heat generation rate ($\propto I^2R$)
R	Internal resistance
θ	State of Health ($\theta = 1$: new, $\theta \approx 0.8$: end-of-life)
$\mathscr{I}_\theta$	Health-dependent fault time intervals
y_{safe}	Safe composite safety assessment
y_{caution}	Caution-level composite safety assessment
y_{critical}	Critical composite safety assessment
SOC	State of Charge
SOH	State of Health
BMS	Battery Management System

Chapter 1

Introduction

1.1 Motivation and Challenges in DES Modeling for Cyber-Physical Systems

The safety and reliability of cyber-physical systems (CPS) have become central concerns as embedded computation increasingly closes the loop with physical processes [1]. CPS couple discrete decision making with the continuous evolution of physical states. This coupling makes correctness inherently time-sensitive: not only *what* happens but also *when* it happens and *for how long* it persists are decisive for both safety and security. Foundational accounts of CPS engineering and standardization emphasize clocks, latency, and synchronization as first-class design dimensions, noting their tight connections to assurance objectives [2], [3].

From a security standpoint, time itself is an information channel. An external observer can glean sensitive facts from timestamps, inter-arrival patterns, or schedule regularity even without internal visibility. Schedule-based side-channel attacks in real-time systems exemplify this risk and motivate defenses that make observable schedules indistinguishable to adversaries [4], [5]. From a safety and reliability standpoint, many abnormal behaviors are *dwell-time-triggered* rather than instantaneous. Phenomena such as thermal runaway, actuator wear, or fatigue typically require the system to remain in vulnerable configurations long enough before faults manifest. Classic results on average dwell time in switched systems formalize this observation [6].

A second observation from practice is that observable outputs are often aggregated and piecewise-constant. Sampling-and-hold interfaces and DAC/ADC conversions naturally produce staircase-like signals. Many instrumentation layers expose binned readouts such as alarm levels, temperature bands, or power steps. For instance, battery management systems report discrete state-of-charge bands (“high,” “medium,” “low”) rather than

continuous voltage values.

The ubiquitous zero-order hold (ZOH) mechanism makes "piecewise constant between sampling instants" a faithful description of how outputs are produced and measured in CPS, not merely an artifact of modeling [7]. Consequently, operators or adversaries reason about *which output label is visible and how long it stays* at that label, rather than about internal events or continuous-valued trajectories.

To analyze such systems at scale, discrete-event systems (DES) provide a mature modeling and algorithmic foundation. DES capture how systems transition between configurations in response to events. They underpin well-developed results on reachability, controllability/observability, diagnosis, and supervisory control, with applications across telecommunication, manufacturing, transportation, and networked systems [8]–[10]. These formalisms have enabled tractable analysis of systems with thousands of states and have been successfully deployed in industrial settings. Despite these strengths, using DES as the primary abstraction for CPS reveals structural mismatches in three fundamental aspects.

First, classical DES are time-abstract by design. Within-state timing effects—for instance, that an output remains at a constant level for a duration before switching—must be emulated by artificially unfolding time into additional states or by injecting ad hoc timer events. These workarounds inflate models, blur the physical meaning of states, and degrade the scalability of verification algorithms. (In the original untimed DES theory, events have no inherent duration [11].)

Second, observability in CPS is partial and aggregated. When only step-wise outputs and their dwell times are visible, pure DES encodings struggle to represent "staying" behaviors without heavy instrumentation. Traditional DES diagnosis frameworks often presuppose richer event visibility than fielded CPS actually provide (e.g. assuming directly observable fault events or high-granularity sensors beyond simple on/off or threshold signals).

Third, many properties of interest are explicitly time- and dwell-dependent. For security, opacity hinges on what an external observer can deduce from output *durations* and inter-switch intervals. For reliability, residence-time conditions frequently specify fault occurrences (e.g. a component overheating only if a high-temperature state persists beyond a threshold duration). Treating time implicitly obscures exactly the triggers and observables that matter in practice.

One may turn to timed and hybrid models to address these gaps. Timed automata and hybrid automata explicitly include clocks and continuous dynamics, respectively, to capture time-sensitive behavior [12], [13]. However, these models face fundamental computational barriers. Dense-time semantics lead to undecidability for timed opacity in timed

automata in general [14]. Attempts to regain decidability through model restrictions—a bounded number of clocks, discrete-time semantics, or constrained observability—either sacrifice expressiveness or still incur prohibitive complexity even for moderately sized systems (e.g. state-based opacity verification for even simple real-time automata is 2-EXPTIME-complete [15]). These challenges motivate the search for alternative abstractions and analysis techniques that can blend the scalability of DES with explicit time semantics.

To bridge these gaps while maintaining computational tractability, we develop the Switching Output Automaton (SOA) framework. The model is built around two core features that directly address the observability and timing challenges identified above. First, SOA enforces a minimal separation between any two consecutive system events—whether state transitions or output changes—through a uniform lower bound on dwell durations. This physically grounded constraint bounds the frequency of observable changes, enabling finite-state verification methods. It reflects the inherent stabilization delays in sensors, actuators, and physical processes. Second, SOA associates each discrete state with a set of possible output values rather than a single output. This set-valued mapping creates structural observational ambiguity: multiple internal states may produce identical outputs, and this ambiguity persists regardless of how precisely timing information is measured.

Within this framework, we focus on two core security and reliability problems: *opacity verification* and *fault diagnosis*. We address both untimed and timed settings. In the untimed setting, opacity verification determines whether external observers can infer secret states from observable event sequences, while fault diagnosis detects faults from partially observed system behaviors. In the timed setting, opacity verification extends to secrets that depend on dwelling in sensitive states for specified durations, and fault diagnosis addresses faults that are enabled only during specific dwell-time windows. Both problems require analysis under partial and aggregated observability, where only piecewise-constant outputs and their durations are visible.

Opacity verification and fault diagnosis are fundamentally inference problems: they characterize what an external observer can deduce from available measurements. In many cyber-physical systems, the measurements are not labeled events but piecewise-constant sensor outputs together with their persistence times. SOA is designed to model exactly this information: states are associated with sets of admissible outputs, outputs may switch without mode changes, and a minimal dwell time bounds the frequency of observable changes. This modeling choice provides both (i) observational ambiguity at the output level—crucial for opacity, since multiple internal states may produce identical outputs—and (ii) a uniform time granularity that enables finite-state abstractions and

observer/diagnoser constructions for timing-aware opacity verification (Chapter 4) and dwell-time-triggered fault diagnosis (Chapter 5).

These problems have been extensively studied in both untimed and timed settings using finite automata, timed automata, Petri nets, and hybrid system formalisms. Before presenting our SOA-based approach and its contributions, we first survey existing work on opacity verification and fault diagnosis to position our contributions and identify the technical gaps that motivate our approach.

1.2 Related Work

1.2.1 Opacity Verification in Discrete Event Systems

Opacity is a security property concerned with whether an external observer (intruder) can infer the occurrence of some secret behavior or state in the system. Formally, a system is opaque if for every execution trace that involves the secret, there exists an indistinguishable trace (with respect to the intruder's observations) that does not involve the secret. We review opacity verification in both untimed and timed settings.

1.2.1.1 Untimed Opacity Verification

Formulations and Verification Methods. In the control systems community, Saboori and Hadjicostis [16] laid the groundwork for state-based opacity and observer-based verification in discrete event systems, introducing initial-state opacity verification (see also [17]). This framework was later extended to K-step and infinite-step opacity with associated complexity analyses [18], [19]. The fundamental idea is to construct an observer (state estimator) that tracks the set of possible system states consistent with each observation sequence, then check whether the observer can ever definitively conclude that the system is in a secret state.

The untimed DES literature has expanded along several fronts. For language-based opacity, canonical formulations and comparisons appear in [20]–[23], with Petri-net based verifiers and linear-constraint formulations in [24], [25]. For state-based variants, initial-state opacity was refined with estimator constructions [17], while current-state opacity has been addressed extensively for Petri net models, including online verification via ILP and verifier-net based methods [26]–[29]. For K-step and infinite-step opacity, early estimator-based algorithms [18], [19] were notably improved by the two-way observer approach [30], with subsequent refinements and complexity characterizations [31], [32].

Complexity and Enforcement. Complexity-wise, opacity verification is PSPACE-complete for several standard formulations and becomes EXPSPACE-complete in modular settings [23], [33], [34]. Beyond verification, opacity enforcement has been pursued via supervisory control [35]–[37], dynamic masks (sensor activation) [33], [38], and insertion/edit functions [39]–[41]. Comprehensive surveys and language-based unifications can be found in [42]–[44].

While untimed opacity provides a foundation for security analysis, it cannot capture timing-based information leakage where the duration of observable behaviors reveals secrets. This limitation motivates the timed extensions reviewed below.

1.2.1.2 Timed Opacity Verification

Challenges and Undecidability. As attention turned to real-time systems, the concept of timed opacity emerged as an extension of opacity to timed DES, where timestamps or inter-event delays are relevant. In a timed setting, an intruder may observe not only a sequence of visible events, but also the timestamps or time differences between events. This opens a new side-channel: even if the sequence of observable actions does not reveal a secret, the timing of those actions might. Timed opacity requires that the intruder, even when exploiting timing information, cannot definitively infer the secret.

Introducing time severely increases complexity. A seminal result by Cassez [14]—"The Dark Side of Timed Opacity"—proved that checking opacity in a timed automaton is undecidable in general. Strong negative results persist even for several restricted classes (e.g., deterministic timed automata). This negative result highlighted the "dark side" of adding time: unlike opacity in finite-state DES (which is decidable in PSPACE), timed opacity verification is, in general, an intractable problem due to the automaton's infinite state space stemming from real-valued clocks. This discovery spurred researchers to explore special cases and alternative problem formulations under which timed opacity can be analyzed effectively.

Decidable Subclasses. One fruitful approach to regain decidability is to constrain the timed automaton's structure or clock features. Wang, Zhan, and An [45] introduced real-time automata (RTA) as the first decidable subclass for opacity analysis, imposing restrictions on clock resets and constraints to enable reduction to finite-state models. Zhang [15], [46] showed that verifying state-based opacity in RTA is 2-EXPTIME-complete and refined verification procedures using observer graphs and trace equivalence.

Another decidable model is the constant-time labeled automaton (CTLA), studied by Li et al. In a CTLA, each event has a fixed (constant) duration, which simplifies the timing dynamics. Li, Lefebvre, Hadjicostis, *et al.* [47] developed an observer-based method using elapsed-time graphs to perform state estimation under timing constraints. Li, Lefebvre, Hadjicostis, *et al.* [48] introduced two notions of state-based timed opacity for CTLAs and provided verification algorithms for both. By restricting all events to constant durations, the infinite state space collapses into a finite progression, making opacity checkable with adapted DES verification techniques.

Recent work [49], [50] has shown that timed automata interpreted over discrete time (integer time semantics) admit decidable opacity verification. The key technique is to construct a tick-equivalent automaton where a special "tick" event represents discrete time elapse, converting the problem to a classical DES opacity check on a finite-state abstraction.

Weakening Intruder Capabilities. Another line of inquiry maintains a general timed system model but limits what the intruder can observe, recovering decidability by defining restricted observation models. André, Lime, Marinho, *et al.* [51] introduced execution-time opacity, where the intruder only measures total execution time rather than individual event timestamps, and showed this is decidable for timed automata. Ammar, Touati, Yeddes, *et al.* [52] formalized bounded-time opacity, requiring secrets remain hidden only for a finite duration, which sidesteps undecidability by bounding the observation horizon. André, Dépernet, and Lefaucheux [53] showed that limiting the intruder to a finite number N of observations makes opacity decidable for all timed automata. By weakening intruders—restricting their time precision, observation length, or aggregation capabilities—researchers have formulated decidable timed opacity problems that often match practical scenarios of limited observation.

Verification Techniques. Verifying timed opacity typically involves constructing an abstraction of what an intruder can infer. Observer-based verification is a common thread in many approaches. In untimed DES, one constructs a knowledge estimator (or observer automaton) that represents all possible states the intruder thinks the system could be in, given the observations so far. This idea carries into timed systems: one constructs a timed observer (often as a product of the system with a copy of itself to compare secret vs non-secret runs). Because of the dense time continuum, constructing an exact timed observer can be complex or infinite; thus, methods employ region graphs or symbolic states (zones). For example, Wang, Zhan, and An [45] used trace-equivalence and region graph techniques to reduce the opacity of an RTA to a language inclusion problem over finite

automata. Li, Lefebvre, Hadjicostis, *et al.* [48] utilize elapsed-time graphs (a zone-based state-space representation) to perform state estimation for CTLAs. In verifying discrete-time opacity [49], [50], the construction of a tick automaton converts the problem to a classical DES opacity check.

1.2.2 Fault Diagnosis in Discrete Event Systems

Fault diagnosis is the task of detecting and isolating fault occurrences by observing system behaviors under partial observability. We review fault diagnosis in both untimed and timed settings, covering modeling formalisms, diagnostic algorithms, and diagnosability notions.

1.2.2.1 Untimed Fault Diagnosis: Diagnosers and Diagnosability

Classic DES diagnosis frameworks model plants as finite-state automata and rely on diagnosers that infer fault occurrence from partially observed event sequences. The seminal work by Sampath, Sengupta, Lafortune, *et al.* [10] introduced the diagnoser automaton construction: a deterministic observer that tracks both system states and fault information, labeling each observation sequence as normal (N), faulty (F), or uncertain (U). A system is diagnosable if every fault can be detected within a finite number of observable events after its occurrence.

Subsequent work refined these foundations along several directions. Polynomial-time diagnosability tests and complexity characterizations appear in [54]–[57]. Safety-critical and active diagnosis variants were developed in [58], [59]. Decentralized and codiagnosis for multi-observer settings were addressed in [60]–[63]. Petri net realizations and online implementation-oriented methods appear in [64]–[69]. Robustness to observation losses and sensor failures was studied in [70]–[74]. Comprehensive surveys of untimed DES diagnosis can be found in [75].

While untimed diagnosis provides efficient detection of event-based faults, it cannot handle faults that are triggered by dwelling in specific states for certain durations or that exhibit temporal patterns. This limitation motivates timed extensions.

1.2.2.2 Timed Fault Diagnosis: Modeling and Verification

Timed fault diagnosis extends untimed frameworks by incorporating timing constraints and temporal information, which is crucial for time-critical systems where the timing of events affects fault manifestation. By leveraging timestamps and time bounds on events, timed fault diagnosis can distinguish faults that would be ambiguous in an untimed setting

and can potentially detect faults more quickly (within a bounded delay) by exploiting the expected temporal patterns of normal vs. faulty behavior. We review the main modeling formalisms and verification methods.

Timed Automata. A widely used model for timed DES is the timed automaton, introduced by Alur and Dill [12] as an automaton extended with real-valued clocks to impose timing constraints on state transitions. Timed automata provide a natural way to model event scheduling and timeouts. In fault diagnosis contexts, the system is modeled as a timed automaton (or a network of timed automata), where faults are modeled either as unobservable fault events or as violations of timing constraints. For example, Tripakis [76] was among the first to introduce a formal framework for fault diagnosis using timed automata, formulating diagnosability for real-time DES and proposing an algorithm based on exploring the zone graph (a symbolic state-space of clock zones) to track possible timed behaviors. Subsequent work by Bouyer, Chevalier, and D'Souza [77] developed a deterministic timed automaton diagnoser by constructing a product of the plant model with observer clocks, converting the problem to a standard language inclusion check. In these approaches, timing constraints help refine the estimation of the system's state: the diagnoser tracks not only which states or faults are possible given the observations, but also the possible clock valuations (time elapsed since certain events). This model-based approach using timed automata has been applied to systems such as communication protocols and real-time controllers, where capturing time between events is essential for distinguishing faults.

Timed Petri Nets. Time Petri nets (TPNs) are another prominent modeling tool for timed DES. Petri nets naturally handle concurrency and distributed events, and time Petri nets attach time intervals to transitions to model delays. In timed fault diagnosis, a labeled time Petri net can model the nominal and faulty behavior of a plant, with some transitions designated as fault occurrences. Several works have extended classic Petri net diagnosis techniques (which estimate markings or sequences from partial observations) to the timed case. For instance, Ghazel, Toguyéni, and Yim [78] formulated diagnosability for labeled time Petri net systems, showing how to verify if any fault transition inevitably leads to an observable pattern (within bounded time) that distinguishes it from all non-faulty executions. Their approach uses the construction of a state class graph (also known as a timed reachability graph) which enumerates reachable markings along with clock constraints, analogous to the zone graph of timed automata, to analyze fault distinguishability. An important notion introduced in Petri net models is K-diagnosability, meaning a

fault can be detected within K observable events after it occurs. This concept can translate to a time-bounded diagnosability if one relates K events to an upper-bound on time. Other Petri net-based modeling variants include timed event graphs (a subclass of Petri nets suitable for cyclic processes with timing) and time Petri nets with priorities used in chronicle-based diagnosis. These models allow incorporating scheduling decisions and resource conflicts into the diagnosis model. Overall, timed Petri nets are particularly useful in concurrent DES diagnosis, where multiple sequences of events (possibly in parallel components) and their timing must be considered.

Other Modeling Frameworks. Beyond timed automata and Petri nets, researchers have explored specialized timed DES formalisms. Time-interval automata (or TIDES) constrain events with intervals and have been used to model systems where only certain timed event sequences are feasible; diagnosability analysis in such models leverages time bounds to improve diagnostic accuracy. Durational graphs (timed automata restricted to single-clock durations) have been combined with algebraic methods (dioid algebra) to efficiently represent concurrent timed behaviors. Hybrid models are also notable: for example, durational transition graphs or timed failure propagators abstract continuous dynamics into timed events. In summary, a rich variety of modeling techniques exist, but the choice of formalism typically depends on the system characteristics: timed automata are common for their formal precision and available verification tools, while timed Petri nets offer clarity for concurrent processes and have inspired efficient analysis algorithms (e.g., state class methods).

Diagnoser Automata and State-Space Exploration. Many timed diagnosis algorithms extend the diagnoser approach introduced by Sampath, Sengupta, Lafortune, *et al.* [10] for untimed DES, where a diagnoser automaton is built to track the information state (the set of possible plant states given observations) and to flag fault occurrences when they become certain. In timed systems, constructing such a diagnoser is more complex because one must track timing information as well. Tripakis [76] proposed an online algorithm that explores the zone-based state space of the timed automaton model while reading observations. Each diagnoser state is a set of plant states combined with a zone (constraints on clock values); by advancing time and events, the diagnoser computes reachable zones and identifies whether a fault has definitely occurred. Bouyer, Chevalier, and D'Souza [77] showed how to synthesize a deterministic observer (diagnoser) for timed automata by a refined powerset construction on the region graph. Their diagnoser is itself a timed automaton that recognizes precisely the timed observation language of the faulty behavior, achieving online fault detection as a timed language acceptance problem. Another

influential strategy was introduced by Hashtrudi Zad, Kwong, and Wonham [79], who proposed incorporating a global clock "tick" event into the model to discretize time. By modeling the passage of each time unit as an observable event, they reduced the timed diagnosis problem to a discrete (untimed) event diagnosis problem in an augmented system. This approach allowed the use of existing untimed diagnoser algorithms on the tick-augmented model, effectively encoding timing information in the event sequence. However, the addition of tick events can cause state-space explosion if the time horizon is large.

For time Petri net models, analogous techniques exist. Ghazel, Toguyéni, and Yim [78] reduce diagnosability verification to a search problem on the timed reachability graph, checking for indistinguishable sequences that violate diagnosability (much like the twin-plant method in untimed DES). On-line algorithms for time Petri nets incrementally update the set of possible markings and clock valuations as observations arrive, exploiting the state class graph to avoid enumerating infinitely many timed states. The complexity of state-space exploration algorithms is generally high (timed diagnoser construction is often of exponential complexity and in some cases diagnosability verification for timed models is PSPACE-complete). Therefore, research continues on improving efficiency, e.g., by using efficient data structures (zones, DBMs), clustering of states, or restricting to subclasses like single-clock automata or bounded intervals where polynomial algorithms exist.

Observer-Based and Analytical Methods. In parallel with automata-based diagnosers, observer-based approaches leverage analytical or structural properties to detect faults without constructing the full state space. These methods are akin to state observers in control theory—they use the model to compute expected observations or system states in real-time and compare with actual observations to indicate faults. Puig, Ocampo-Martínez, Pérez, *et al.* [80] introduced interval observers for timed DES diagnosis, where an interval observer maintains upper and lower bounds on timed behavioral trajectories. Discrepancies between the observer-predicted time bounds and the real observations signal faults. This approach, demonstrated on water distribution and sewer network control, shows that integrating temporal logic with observer theory can improve fault isolation in systems with both logical events and time-driven processes. Algebraic Petri net observers can also be designed to estimate the marking (state) of a Petri net from partial event observations. By maintaining an estimate of possible token counts and transition firings over time, such observers can detect when the system's behavior deviates from the model's prediction. These observers often rely on solving systems of linear equations or

inequalities that describe the Petri net's token flow with timing constraints, and they can be computationally lighter than building the full diagnoser reachability graph.

Another line of work uses chronicles and patterns to perform diagnosis. A chronicle is a temporally annotated event sequence representing a typical fault scenario. Pencolé and Subias [81] developed a chronicle-based diagnosability framework for timed systems, where known fault patterns (chronicles) are matched against the observed behavior in real-time. Each chronicle encodes a temporal pattern (e.g. event A occurs, then within 5–7 seconds event B occurs, etc.) that corresponds to a fault. Diagnosability is analyzed by checking that no single observation sequence can satisfy both a fault chronicle and a normal behavior chronicle. This approach leans on AI techniques for pattern recognition and was applied to web services and workflows. Pattern-based methods can be seen as knowledge-based diagnosis: they rely on pre-specified fault signatures rather than exhaustively analyzing the entire state space, and are useful when certain faults have well-characterized temporal signatures.

Online vs. Offline Diagnosis. A key distinction in diagnosis strategies is online vs. offline processing. Online diagnosis monitors the system in real-time and raises alarms as soon as enough evidence of a fault is observed. The diagnoser automaton reads events as they happen and may identify a fault after a certain delay (the diagnosability condition ensures this delay is finite). For example, an online timed diagnoser might use clock constraints to determine that "if event E did not occur within 10 seconds, fault F must have happened." Offline diagnosis usually refers to post-mortem analysis or to algorithms that, given a recorded observation sequence, determine which fault happened and when. Offline methods are often used for diagnosability verification—checking from the model whether faults would be detectable in principle. Many timed diagnosability verification algorithms [64], [76] are offline computations performed on the model. They answer questions like: Is the system diagnosable? and Within what bound will a fault always be detected?

An important metric is the guaranteed detection latency. This relates to time-bounded diagnosability: a system is Δ-diagnosable if any fault will be detected within time Δ of its occurrence. In practice, hybrid approaches are common: one first performs offline analysis on the model to verify diagnosability and perhaps synthesize a diagnoser, and then implements the diagnoser for online monitoring.

1.3 Contributions

This thesis addresses security and reliability analysis for cyber-physical systems under partial and aggregated observability, where time fundamentally governs both what can be revealed to adversaries and when faults become physically realizable. As surveyed in Section 1.2, existing approaches using dense-time models face undecidability or prohibitive complexity, while classical discrete-event systems require artificial augmentation to represent timing and piecewise-constant outputs.

We introduce the Switching Output Automaton (SOA) framework to bridge this gap. SOA models what observers see in practice: output symbols and their durations. The key innovation is a minimal dwell time constraint, which requires that any two consecutive events—whether state transitions or output switches—be separated by at least a fixed positive duration. This constraint reflects physical reality (sensors, actuators, and processes inherently require non-zero stabilization intervals) and enables decidable verification by bounding the rate of observable changes. Outputs can switch while the system remains in a single state, and timing information is built into the model structure rather than simulated through auxiliary mechanisms.

Main contributions. This thesis makes the following contributions:

- **Switching Output Automaton framework.** We formalize SOA as a continuous-time discrete-event model where states are associated with sets of possible outputs, outputs can switch within states without requiring state transitions, and any two consecutive changes are separated by the minimal dwell time. We construct the evolution automaton that abstracts continuous-time behavior into discrete logical states by tracking dwell-time satisfaction, and develop the observer automaton for deterministic state estimation through subset construction and state simplification.

- **Opacity verification.** We establish decidable verification procedures for both current-state opacity (where secrets are discrete states) and timed opacity (where secrets are global-state-duration triples (x, y, t) combining discrete state, output, and dwell time). For timed opacity, we introduce secret-dependent time discretization that applies fine-grained quantization only to vulnerable global states, reducing verification to finite-state reachability analysis in the observer automaton. We prove that timed opacity holds if and only if no reachable observer state consists entirely of secret logical states.

- **Timed fault diagnosis.** We extend SOA to the Switching Output Automaton with Faults (SOAF) framework that models faults enabled only during specific dwell-time windows in fault-prone states. We develop the diagnoser construction through time discretization, evolution automaton synthesis, fault recognizer construction, and observer-based determinization. The resulting diagnoser provides online classification of observation sequences as normal, faulty, or uncertain based on dwell-time-triggered fault semantics.

- **Practical validation.** We demonstrate the practical applicability of the SOA framework through three comprehensive case studies: current-state opacity verification for a smart water supply system under aggregate power consumption monitoring, timed opacity verification for a patient monitoring system with healthcare privacy requirements, and timed fault diagnosis for a battery management system detecting thermal runaway conditions.

- **Open-source implementation.** To support reproducibility and further research, we provide SOA Toolbox, an open-source implementation of the verification and diagnosis framework available on GitHub.[1]

1.4 Thesis Outline

The remainder of this thesis is organized as follows:

- **Chapter 2 (Preliminaries).** We introduce fundamental concepts from discrete event systems and automata theory, including deterministic and nondeterministic finite automata, observer construction for state estimation under partial observability, current-state opacity verification, and fault diagnosis in discrete event systems. These concepts establish the theoretical foundation for the SOA framework developed in subsequent chapters.

- **Chapter 3 (Switching Output Automaton Framework).** We formally define the Switching Output Automaton model, including its syntax (states, outputs, arcs, minimal dwell time), semantics (state-output runs, output behaviors), and observations. We construct the evolution automaton as a finite-state abstraction of the continuous-time SOA behavior, and develop the observer automaton for deterministic state estimation. This chapter provides the core modeling framework used throughout the thesis.

[1] https://github.com/ty451/Switching-Output-Automata

- **Chapter 4 (Opacity Verification in Switching Output Automata).** We address both current-state opacity and timed opacity verification for SOA. For timed opacity, we introduce the concept of vulnerable global states, develop secret-dependent time discretization, construct the secret-dependent evolution automaton, and establish the verification condition based on observer reachability analysis. We present the complete verification algorithm and analyze its computational complexity.

- **Chapter 5 (Timed Fault Diagnosis in Switching Output Automata).** We extend SOA to SOAF by incorporating fault arcs with time-dependent enablement windows. We develop a four-stage fault diagnosis pipeline: logical state discretization, evolution automaton with faults construction, fault recognizer synthesis, and diagnoser construction. The diagnoser provides online fault detection and classification capabilities based on observed output behaviors.

- **Chapter 6 (Case Studies).** We validate the practical applicability of the SOA framework through three case studies from different cyber-physical system domains: (i) a smart water supply system verifying current-state opacity under power monitoring, (ii) a patient monitoring system verifying timed opacity for healthcare privacy, and (iii) a battery management system performing timed fault diagnosis for thermal runaway detection. Each case study demonstrates the modeling approach, verification procedure, and analysis results.

- **Chapter 7 (Conclusion and Future Work).** We summarize the main contributions of this dissertation, discuss the advantages and limitations of the SOA framework, and outline promising directions for future research including scalability improvements, stochastic extensions, multi-agent systems, and integration with continuous control.

Chapter 2

Preliminaries

This chapter introduces the fundamental concepts from discrete event systems and automata theory that underpin the frameworks developed in this dissertation. Section 2.1 establishes the basic notation for formal languages and deterministic finite automata. Section 2.2 introduces nondeterministic finite automata and the observer construction for state estimation under partial observability. Section 2.3 defines current-state opacity and its verification procedure. Section 2.4 presents fault diagnosis in discrete event systems using the diagnoser approach.

2.1 Formal Languages and Deterministic Finite Automata

An *alphabet* Σ is a finite and non-empty set of symbols. The number of symbols it contains is called its *cardinality* and is denoted by $|\Sigma|$. A *word* (or *string* or *trace*) w over an alphabet Σ is a finite sequence of symbols from Σ. The number of symbols in the word is called its *length* and is denoted by $|w|$, and $|w|_\sigma$ denotes the number of occurrences of symbol $\sigma \in \Sigma$ in w.

The set of all words over an alphabet Σ is denoted by Σ^*, known as the *Kleene closure* (or *Kleene star*) of Σ. The *empty word* ε (the sequence of zero length) belongs to Σ^* for every alphabet Σ. While an alphabet Σ is finite, the set Σ^* is always infinite. This notation extends naturally to languages: for any language L, the Kleene star L^* denotes the set of all words formed by concatenating zero or more words from L (formally defined below).

The *concatenation* of two words $w_1, w_2 \in \Sigma^*$ is the word $w = w_1 \cdot w_2 \in \Sigma^*$ consisting of the sequence of symbols in w_1 followed by the sequence of symbols in w_2. Concatenation is associative but not commutative, and the empty word ε serves as the identity element.

If a word $w \in \Sigma^*$ can be written as $w = uvz$ where $u, v, z \in \Sigma^*$, then u is called a *prefix* of w (denoted $u \preceq w$), v is called a *substring* of w, and z is called a *suffix* of w. Given a

word $w \in \Sigma^*$ and a subset alphabet $\hat{\Sigma} \subseteq \Sigma$, the *projection* of w on $\hat{\Sigma}$, denoted by $w \uparrow \hat{\Sigma}$, is the word obtained by removing from w all symbols that do not belong to $\hat{\Sigma}$.

A *language* L over an alphabet Σ is a set of words over this alphabet, with cardinality $|L|$ denoting the number of words it contains; a language can be empty ($\emptyset$), finite, or infinite. For two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ (with $\bar{\Sigma} = \Sigma_1 \cap \Sigma_2$ and $\Sigma = \Sigma_1 \cup \Sigma_2$), the following operators are defined:

- *Union*: $L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \vee w \in L_2\}$.
- *Intersection*: $L_1 \cap L_2 = \{w \in \bar{\Sigma}^* \mid w \in L_1, w \in L_2\}$.
- *Concatenation*: $L_1 L_2 = \{w = w_1 w_2 \in \Sigma^* \mid w_1 \in L_1, w_2 \in L_2\}$.
- *Kleene star*: $L^* = \{\varepsilon\} \cup L \cup LL \cup LLL \cup \cdots = \bigcup_{k=0}^{\infty} L^k$.
- *Prefix closure*: $\text{pref}(L) = \{u \in \Sigma^* \mid \exists w \in L : u \preceq w\}$; a language L is prefix closed if $L = \text{pref}(L)$.
- *Complement*: $\bar{L} = \{w \in \Sigma^* \mid w \notin L\} = \Sigma^* \setminus L$.
- *Concurrent composition*: $L_1 \| L_2 = \{w \in \Sigma^* \mid w \uparrow \Sigma_1 \in L_1, w \uparrow \Sigma_2 \in L_2\}$; when $\Sigma_1 = \Sigma_2 = \Sigma$, this reduces to intersection: $L_1 \| L_2 = L_1 \cap L_2$.

Example 2.1 (Formal Languages)**.** *Consider the alphabet $\Sigma = \{a, b, c\}$ with cardinality $|\Sigma| = 3$. The word $w_1 = abbc$ over Σ has length $|w_1| = 4$, with $|w_1|_a = 1$, $|w_1|_b = 2$, and $|w_1|_c = 1$. The set Σ^* contains all finite sequences of symbols from Σ, including ε (length 0), single symbols a, b, c (length 1), and longer words such as $aa, ab, ac, ba, bb, bc, ca, cb, cc$ (length 2). Concatenating $w_2 = ab$ with $w_3 = bc$ yields $w_2 \cdot w_3 = abbc = w_1$, with $|w_2 \cdot w_3| = |w_2| + |w_3| = 2 + 2 = 4$. Note that concatenation is not commutative: $w_3 \cdot w_2 = bcab \neq abbc = w_2 \cdot w_3$.*

The prefixes of $w_1 = abbc$ are $\varepsilon, a, ab, abb, abbc$; its suffixes are $\varepsilon, c, bc, bbc, abbc$; and its substrings include all prefixes, all suffixes, plus b, bb. For the projection operator, let $\hat{\Sigma} = \{a, b\} \subseteq \Sigma$: then $w_1 \uparrow \hat{\Sigma} = abb$ (obtained by removing the symbol c).

Consider now the languages $L_1 = \{a, ab, abb\}$, $L_2 = \{b, bc\}$, and $L_3 = \{ab^n \mid n \geq 0\} = \{a, ab, abb, abbb, \ldots\}$. Language L_1 has cardinality $|L_1| = 3$, while L_3 is infinite with $|L_3| = \infty$. The union $L_1 \cup L_2 = \{a, ab, abb, b, bc\}$ and intersection $L_1 \cap L_3 = \{a, ab, abb\} = L_1$ (note that $L_1 \subseteq L_3$). The concatenation $L_1 L_2 = \{ab, abc, abb, abbc, abbb, abbbc\}$ consists of six words. The Kleene star $L_2^* = \{\varepsilon\} \cup \{b, bc\} \cup \{bb, bbc, bcb, bcbc\} \cup \cdots$ contains all words formed by concatenating elements of L_2 any number of times.

The prefix closure $\text{pref}(L_1) = \{\varepsilon, a, ab, abb\} \supsetneq L_1$*, so* L_1 *is not prefix closed, whereas* $\text{pref}(L_2) = \{\varepsilon, b, bc\} \supsetneq L_2$*, but* $\text{pref}(\text{pref}(L_2)) = \text{pref}(L_2)$*. The complement of* L_1 *over alphabet* Σ *is* $\bar{L}_1 = \Sigma^* \setminus L_1$*, containing all words except* a, ab, abb*. For concurrent composition, let* $\Sigma_1 = \{a,b\}$*,* $\Sigma_2 = \{b,c\}$*,* $L_4 = \{ab^n \mid n \geq 0\} \subseteq \Sigma_1^*$*, and* $L_5 = \{cbc^n b \mid n \geq 0\} \subseteq \Sigma_2^*$*. Then* $L_4 \| L_5 = \{acbc^n b \mid n \geq 0\} \cup \{cabc^n b \mid n \geq 0\}$ *over alphabet* $\Sigma_1 \cup \Sigma_2 = \{a,b,c\}$*. For example, word* $w = acbcb$ *belongs to* $L_4 \| L_5$ *because* $w \uparrow \Sigma_1 = abb \in L_4$ *and* $w \uparrow \Sigma_2 = cbcb \in L_5$*.*

Definition 2.1 (Deterministic Finite Automaton)**.** *A* deterministic finite automaton *(DFA)* $G = (X, \Sigma, \delta, x_0, X_m)$ *is a five-tuple consisting of a finite set of states* X*, an alphabet* Σ*, a (partial) transition function* $\delta : X \times \Sigma \to X$*, an initial state* $x_0 \in X$*, and a set of final (or marked) states* $X_m \subseteq X$*.*

The transition function specifies the system dynamics: if $\bar{x} = \delta(x, \sigma)$, then the occurrence of event σ when the current state is x leads to state $\bar{x}$. A *run* is a sequence of transitions $x_0 \xrightarrow{\sigma_1} x_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} x_n$ where each transition $x_{i-1} \xrightarrow{\sigma_i} x_i$ satisfies $\delta(x_{i-1}, \sigma_i) = x_i$ for $i = 1, \ldots, n$, and the word generated by this run is $w = \sigma_1 \sigma_2 \cdots \sigma_n$ (with the convention that the run consisting of a single state with no transitions generates the empty word ε). The *extended transition function* $\delta^* : X \times \Sigma^* \to X$ is defined such that $\delta^*(x, w) = \bar{x}$ if there exists a run from state x to state $\bar{x}$ that generates word w, with $\delta^*(x, \varepsilon) = x$ for all $x \in X$.

A word $w \in \Sigma^*$ is *generated* if $\delta^*(x_0, w)$ is defined (i.e., there exists a run from the initial state producing w) and is *accepted* if $\delta^*(x_0, w) \in X_m$ (i.e., the run reaches a final state). The *generated language* $L(G) = \{w \in \Sigma^* \mid \delta^*(x_0, w) \text{ is defined}\}$ contains all words produced by runs starting from the initial state, while the *accepted language* $L_m(G) = \{w \in \Sigma^* \mid \delta^*(x_0, w) \in X_m\} \subseteq L(G)$ contains only those words whose runs terminate in final states. The generated language is always prefix closed ($L(G) = \text{pref}(L(G))$), whereas the accepted language satisfies $L_m(G) \subseteq \text{pref}(L_m(G)) \subseteq L(G)$.

Definition 2.2 (Concurrent Composition of DFAs)**.** *For two DFAs* $G_1 = (X_1, \Sigma_1, \delta_1, x_{1,0}, X_{1,m})$ *and* $G_2 = (X_2, \Sigma_2, \delta_2, x_{2,0}, X_{2,m})$*, their* concurrent composition *(or* synchronous product*)* $G = G_1 \| G_2$ *is the DFA that generates language* $L(G) = L(G_1) \| L(G_2)$ *and accepts language* $L_m(G) = L_m(G_1) \| L_m(G_2)$*.*

The concurrent composition is constructed with alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$*, initial state* $x_0 = (x_{1,0}, x_{2,0})$*, state space* $X \subseteq X_1 \times X_2$ *containing only reachable states, and final states*

$X_m = X \cap (X_{1,m} \times X_{2,m})$. *The transition function is defined by:*

$$\delta((x_1,x_2),\sigma) = \begin{cases} (\delta_1(x_1,\sigma),x_2) & \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2, \delta_1(x_1,\sigma) \text{ is defined} \\ (x_1,\delta_2(x_2,\sigma)) & \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1, \delta_2(x_2,\sigma) \text{ is defined} \\ (\delta_1(x_1,\sigma),\delta_2(x_2,\sigma)) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2, \\ & \delta_1(x_1,\sigma) \text{ and } \delta_2(x_2,\sigma) \text{ are both defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Importantly, concurrent composition is not limited to DFAs alone: it extends naturally to other finite automaton models including nondeterministic finite automata (NFAs, introduced in Section 2.2), since NFAs and DFAs have equivalent expressive power (both recognize precisely the class of regular languages), and the composition can be performed at the language level or by constructing product automata with appropriate handling of nondeterminism.

Example 2.2 (Deterministic Finite Automata and Concurrent Composition)**.** *Consider two simple DFAs* $G_1 = (X_1,\Sigma_1,\delta_1,x_0,X_{1,m})$ *and* $G_2 = (X_2,\Sigma_2,\delta_2,y_0,X_{2,m})$ *defined as follows.*

DFA G_1 *has alphabet* $\Sigma_1 = \{a,b\}$*, states* $X_1 = \{x_0,x_1\}$*, initial state* x_0*, and final states* $X_{1,m} = \{x_0\}$*. The transition function is:* $\delta_1(x_0,a) = x_1$*,* $\delta_1(x_0,b) = x_0$*,* $\delta_1(x_1,a) = x_0$*,* $\delta_1(x_1,b) = x_1$*. This automaton accepts words containing an even number of a's.*

DFA G_2 *has alphabet* $\Sigma_2 = \{b,c\}$*, states* $X_2 = \{y_0,y_1\}$*, initial state* y_0*, and final states* $X_{2,m} = \{y_0\}$*. The transition function is:* $\delta_2(y_0,b) = y_1$*,* $\delta_2(y_0,c) = y_0$*,* $\delta_2(y_1,b) = y_0$*,* $\delta_2(y_1,c) = y_1$*. This automaton accepts words containing an even number of b's.*

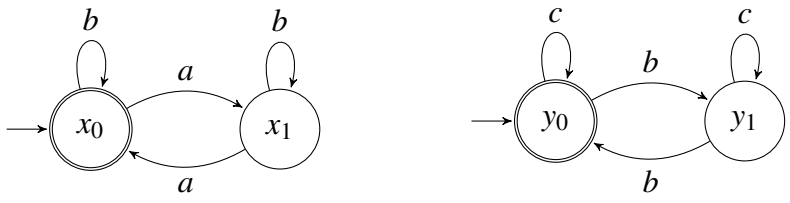


Figure 2.1: DFAs G_1 (left) and G_2 (right).

For the concurrent composition $G = G_1 \| G_2$*, we identify the event types:* $\Sigma_1 \cap \Sigma_2 = \{b\}$ *(synchronized event),* $\Sigma_1 \setminus \Sigma_2 = \{a\}$ *(private to* G_1*), and* $\Sigma_2 \setminus \Sigma_1 = \{c\}$ *(private to* G_2*). The composed alphabet is* $\Sigma = \Sigma_1 \cup \Sigma_2 = \{a,b,c\}$*.*

The state space of G *is* $X = X_1 \times X_2 = \{(x_0,y_0),(x_0,y_1),(x_1,y_0),(x_1,y_1)\}$ *with initial state* (x_0,y_0) *and final states* $X_m = X_{1,m} \times X_{2,m} = \{(x_0,y_0)\}$*. The transition function is*

constructed according to the concurrent composition definition. For example, from state (x_0,y_0)*: event a (private to* G_1*) yields* $\delta((x_0,y_0),a) = (\delta_1(x_0,a),y_0) = (x_1,y_0)$*; event b (synchronized) yields* $\delta((x_0,y_0),b) = (\delta_1(x_0,b),\delta_2(y_0,b)) = (x_0,y_1)$*; event c (private to* G_2*) yields* $\delta((x_0,y_0),c) = (x_0,\delta_2(y_0,c)) = (x_0,y_0)$*.*

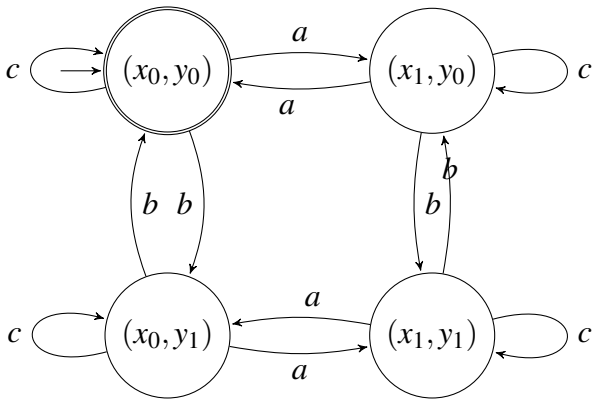


Figure 2.2: Concurrent composition $G = G_1 \| G_2$.

The composed automaton G accepts words over $\{a,b,c\}$ *containing an even number of both a's and b's. For instance, consider the word* $w = abc$*: starting from* (x_0,y_0)*, we have* $(x_0,y_0) \xrightarrow{a} (x_1,y_0) \xrightarrow{b} (x_1,y_1) \xrightarrow{c} (x_1,y_1)$*, but this terminates in* $(x_1,y_1) \notin X_m$*, so* $abc \notin L_m(G)$*. However, the word* $w' = abab$ *is accepted:* $(x_0,y_0) \xrightarrow{a} (x_1,y_0) \xrightarrow{b} (x_1,y_1) \xrightarrow{a} (x_0,y_1) \xrightarrow{b} (x_0,y_0) \in X_m$*, so* $abab \in L_m(G)$*.*

2.2 Nondeterministic Finite Automata and Observer Construction

Definition 2.3 (Nondeterministic Finite Automaton)**.** *A* nondeterministic finite automaton *(NFA)* $G = (X,\Sigma,\Delta,x_0,X_m)$ *is a five-tuple consisting of a finite set of states X, an alphabet* Σ*, a* transition relation $\Delta \subseteq X \times \Sigma_\varepsilon \times X$ *(where* $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$*), an initial state* $x_0 \in X$*, and a set of final (or marked) states* $X_m \subseteq X$*.*

The transition relation Δ generalizes the transition function of DFAs by allowing multiple possibilities: if $(x,\sigma',\bar{x}) \in \Delta$, then from state x the occurrence of a σ'-transition (where $\sigma' \in \Sigma_\varepsilon$) leads to state $\bar{x}$. NFAs capture two distinct modeling aspects:

- *Nondeterminism*: Multiple transitions from the same state with identical labels represent genuine branching in the system dynamics, where the same event may lead to different successor states.

- *Partial observability*: Transitions labeled with ε represent events that are unobservable to external observers, modeling sensor limitations, internal state changes, or hidden fault events. From an observer's perspective, the system may transition without producing any visible output.

A *run* is a sequence $x^{(0)} \xrightarrow{\sigma'_1} x^{(1)} \xrightarrow{\sigma'_2} \cdots \xrightarrow{\sigma'_k} x^{(k)}$ where $(x^{(i-1)}, \sigma'_i, x^{(i)}) \in \Delta$ for all $i = 1, \ldots, k$. The run produces word $w \in \Sigma^*$ obtained by concatenating all non-ε labels from $\sigma'_1, \ldots, \sigma'_k$ (i.e., removing all ε symbols), and reaches state $x^{(k)}$.

The *extended transition relation* $\Delta^* \subseteq X \times \Sigma^* \times X$ generalizes Δ to words: $(x, w, \bar{x}) \in \Delta^*$ if there exists a run from x producing w and reaching $\bar{x}$. By definition, $(x, \varepsilon, x) \in \Delta^*$ for all $x \in X$.

A word $w \in \Sigma^*$ is *generated* if there exists $x \in X$ such that $(x_0, w, x) \in \Delta^*$, and is *accepted* if there exists $x \in X_m$ such that $(x_0, w, x) \in \Delta^*$. The *generated language* $L(G) = \{w \in \Sigma^* \mid \exists x \in X : (x_0, w, x) \in \Delta^*\}$ and *accepted language* $L_m(G) = \{w \in \Sigma^* \mid \exists x \in X_m : (x_0, w, x) \in \Delta^*\}$ satisfy $L_m(G) \subseteq \text{pref}(L_m(G)) \subseteq L(G) = \text{pref}(L(G))$.

For partially observed systems, the alphabet Σ is partitioned into *observable events* Σ_o and *unobservable events* $\Sigma_{uo} = \Sigma \setminus \Sigma_o$. The *natural projection* $P : \Sigma^* \to \Sigma_o^*$ is defined recursively as: $P(\varepsilon) = \varepsilon$, $P(\sigma) = \sigma$ if $\sigma \in \Sigma_o$, $P(\sigma) = \varepsilon$ if $\sigma \in \Sigma_{uo}$, and $P(w\sigma) = P(w)P(\sigma)$ for $w \in \Sigma^*$ and $\sigma \in \Sigma$. The projection P extracts the observable content from any word by removing all unobservable events.

An NFA G represents a system whose state cannot be directly observed. After observing $o = P(w)$ for some executed word w, the *set of states consistent with observation* o is:

$$\mathscr{X}(o) = \{x \in X \mid \exists w \in \Sigma^* : P(w) = o \text{ and } (x_0, w, x) \in \Delta^*\}.$$

Definition 2.4 (Observer for Nondeterministic Finite Automaton)**.** *Given an NFA* $G = (X, \Sigma, \Delta, x_0, X_m)$ *with observable events* Σ_o *and projection* P*, the* observer $\text{Obs}(G) = (Q, \Sigma_o, \delta, q_0, Q_m)$ *is a DFA over the observable alphabet, where:*

- $Q \subseteq 2^X$ *contains only reachable state subsets;*
- $q_0 = \{x \in X \mid (x_0, \varepsilon, x) \in \Delta^*\}$ *(states reachable from* x_0 *via unobservable events);*
- *For* $q \in Q$ *and* $\sigma \in \Sigma_o$, $\delta(q, \sigma) = \{x' \in X \mid \exists x \in q, w \in \Sigma^* : P(w) = \sigma$ *and* $(x, w, x') \in \Delta^*\}$*;*
- $Q_m = \{q \in Q \mid q \cap X_m \neq \emptyset\}$*;*
- *For all* $o \in \Sigma_o^*$, $\mathscr{X}(o) = \delta^*(q_0, o)$.

The observer provides efficient state estimation: for any observed word $o \in \Sigma_o^*$, the state $\delta^*(q_0, o)$ in Obs(G) is exactly the set $\mathscr{X}(o)$ of states consistent with observation o in G. This construction is fundamental for fault diagnosis, opacity verification, and supervisory control under partial observation.

Example 2.3 (NFA and Observer Construction)**.** *Consider an NFA $G = (X, \Sigma, \Delta, x_0, X_m)$ with states $X = \{x_0, x_1, x_2, x_3\}$, alphabet $\Sigma = \{a, b, u\}$, initial state x_0, and final states $X_m = \{x_3\}$. The observable events are $\Sigma_o = \{a, b\}$ and the unobservable event is $u \in \Sigma_{uo}$. The transition relation contains (x_0, u, x_1), (x_0, u, x_2), (x_1, a, x_3), (x_2, a, x_3), and (x_3, b, x_3).*

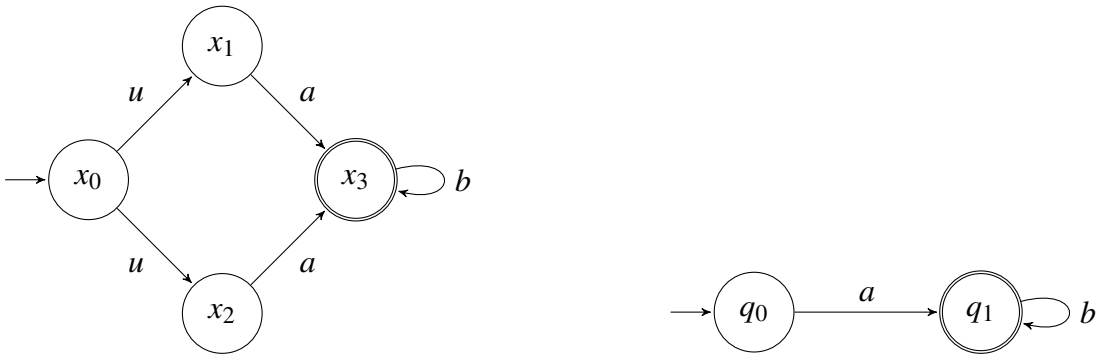


Figure 2.3: NFA G (left) and its observer Obs(G) (right), where $q_0 = \{x_0, x_1, x_2\}$ and $q_1 = \{x_3\}$.

The observer Obs(G) *has two states: $q_0 = \{x_0, x_1, x_2\}$ (initial state) and $q_1 = \{x_3\}$ (final state). The transitions are $\delta(q_0, a) = q_1$ and $\delta(q_1, b) = q_1$. After observing $o = a$, the observer state $q_1 = \{x_3\}$ indicates that the system is in state x_3, though the observer cannot determine whether the system followed the path through x_1 or x_2.*

2.3 Verification of Current-State Opacity

Opacity is an information-flow property that captures the ability of a system to keep certain aspects of its behavior secret from an external observer. In security-critical applications, it is often necessary to ensure that an intruder observing the system's behavior cannot infer whether the system is currently in a secret state.

Consider a system modeled by an NFA $G = (X, \Sigma, \Delta, x_0, X_m)$ with observable events Σ_o and projection $P : \Sigma^* \to \Sigma_o^*$ as defined in Section 2.2. Let $X_s \subseteq X$ denote a set of *secret states* that the system wishes to keep private from an external observer.

Definition 2.5 (Current-State Opacity)**.** *A system G is* current-state opaque *with respect to X_s and P if for every word $w \in L(G)$ such that there exists $x \in X_s$ with $(x_0, w, x) \in \Delta^*$ (i.e., w can lead to a secret state), there exists another word $w' \in L(G)$ with $P(w) = P(w')$*

and $(x_0, w', x') \in \Delta^*$ *for some* $x' \notin X_s$ *(i.e.,* w' *produces the same observation but can lead to a non-secret state).*

Equivalently, a system is current-state opaque if an observer monitoring the observable behavior cannot definitively conclude that the system is currently in a secret state. The observer may suspect that a secret state has been reached, but cannot be certain.

Observer construction provides an efficient mechanism for verifying current-state opacity. The observer $\mathrm{Obs}(G)$ (Definition 2.4) tracks the set of states consistent with each observable word. For any observation $o \in \Sigma_o^*$, the observer state $q = \delta^*(q_0, o)$ represents the set $\mathscr{X}(o)$ of all states consistent with observation o.

Proposition 2.1 (Verification of Current-State Opacity)**.** *A system G is current-state opaque with respect to* X_s *and P if and only if for every reachable state* $q \in Q$ *of the observer* $\mathrm{Obs}(G)$*, the following condition holds:*

$$q \subseteq X_s \implies \mathit{false},$$

i.e., no observer state is entirely contained within the secret states X_s*. Equivalently, for all* $q \in Q$*, either* $q \cap X_s = \emptyset$ *(the observation reveals no secret) or* $q \not\subseteq X_s$ *(the observation is ambiguous).*

The verification procedure thus consists of constructing the observer and checking whether any of its states is a subset of X_s. If such a state exists, then there is an observation that definitively reveals the system is in a secret state, violating opacity. Otherwise, every observation either reveals no secret or maintains ambiguity, ensuring opacity.

Example 2.4 (Verification of Current-State Opacity)**.** *Consider the NFA G from Example 2.3 with secret states* $X_s = \{x_3\}$*. To verify current-state opacity, we examine the observer* $\mathrm{Obs}(G)$ *which has states* $q_0 = \{x_0, x_1, x_2\}$ *and* $q_1 = \{x_3\}$*.*

For each observer state, we check the condition in Proposition 2.1:

- *For* $q_0 = \{x_0, x_1, x_2\}$*: Since* $q_0 \cap X_s = \{x_0, x_1, x_2\} \cap \{x_3\} = \emptyset$*, this state reveals no secret.*
- *For* $q_1 = \{x_3\}$*: Since* $q_1 = \{x_3\} \subseteq X_s$*, this state is entirely contained within the secret states.*

Because $q_1 \subseteq X_s$*, the system is* not *current-state opaque. After observing* $o = a$*, the observer reaches state* q_1*, allowing the intruder to definitively conclude that the system is in the secret state* x_3*. The system violates opacity because the observation a unambiguously reveals the secret.*

2.4 Fault Diagnosis in Discrete Event Systems

Fault diagnosis concerns detecting and identifying faults in systems based on observable behavior. In discrete event systems, faults are modeled as unobservable events, and the diagnosis problem consists of determining whether a fault has occurred based on the sequence of observable events.

Consider a system modeled by a DFA $G = (X, \Sigma, \delta, x_0)$ where the alphabet is partitioned as $\Sigma = \Sigma_o \cup \Sigma_{uo}$, with Σ_o being observable events and Σ_{uo} being unobservable events. The unobservable events are further partitioned as $\Sigma_{uo} = \Sigma_f \cup \Sigma_{reg}$, where Σ_f denotes *fault events* and Σ_{reg} denotes *regular unobservable events* that do not represent faults. Let $P : \Sigma^* \to \Sigma_o^*$ denote the natural projection as defined in Section 2.2.

The *support* of a string $s \in \Sigma^*$ is defined as $||s|| = \{\sigma \in \Sigma \mid |s|_\sigma > 0\}$, the (unordered) set of all events that appear at least once in s (ignoring multiplicity and order).

Definition 2.6 (Diagnosis Function)**.** *Given a DFA G with fault events Σ_f, a* diagnosis function $\varphi : \Sigma_o^* \to \{N, F, U\}$ *associates with each observed word $w \in \Sigma_o^*$ a diagnosis state:*

- $\varphi(w) = N$ *(no fault): if for all $s \in P^{-1}(w)$, $||s|| \cap \Sigma_f = \emptyset$ (no string consistent with w contains a fault event);*
- $\varphi(w) = F$ *(fault): if for all $s \in P^{-1}(w)$, $||s|| \cap \Sigma_f \neq \emptyset$ (all strings consistent with w contain a fault event);*
- $\varphi(w) = U$ *(uncertain): if there exist $s', s'' \in P^{-1}(w)$ such that $||s'|| \cap \Sigma_f = \emptyset$ and $||s''|| \cap \Sigma_f \neq \emptyset$ (the observation is ambiguous).*

The diagnosis function can be computed efficiently using a *diagnoser*, constructed through a three-step process involving the fault monitor, fault recognizer, and observer construction.

Definition 2.7 (Fault Monitor)**.** *Given an alphabet Σ and fault events $\Sigma_f \subseteq \Sigma$, the* fault monitor *is the DFA $M = (X_M, \Sigma, \delta_M, x_{M,0})$ with states $X_M = \{N, F\}$, initial state $x_{M,0} = N$, and transition function:*

$$\delta_M(N, \sigma) = \begin{cases} N & \text{if } \sigma \in \Sigma \setminus \Sigma_f \\ F & \text{if } \sigma \in \Sigma_f \end{cases}, \quad \delta_M(F, \sigma) = F \text{ for all } \sigma \in \Sigma.$$

For any string $s \in \Sigma^*$, $\delta_M^*(N, s) = F$ (where δ_M^* denotes the extended transition function as defined for DFAs) if and only if $||s|| \cap \Sigma_f \neq \emptyset$.

Definition 2.8 (Fault Recognizer)**.** *Given a DFA $G = (X, \Sigma, \delta, x_0)$ with fault events Σ_f and fault monitor M, the* fault recognizer *is* $\mathrm{Rec}(G) = G \| M$*, obtained by concurrent composition. The fault recognizer has states $X_R \subseteq X \times \{N, F\}$ and initial state (x_0, N). For any string $s \in L(G)$ with $\delta^*(x_0, s) = x$, the fault recognizer reaches state (x, N) if $||s|| \cap \Sigma_f = \emptyset$, and state (x, F) if $||s|| \cap \Sigma_f \neq \emptyset$.*

Definition 2.9 (Diagnoser)**.** *Given a DFA G with observable events Σ_o and fault events Σ_f, the* diagnoser *is* $\mathrm{Diag}(G) = \mathrm{Obs}(\mathrm{Rec}(G))$*, the observer of the fault recognizer. The diagnoser is a DFA over Σ_o with states $Y \subseteq 2^{X \times \{N,F\}}$, where each state $y \in Y$ is a set of pairs (x, γ) with $x \in X$ and $\gamma \in \{N, F\}$.*

Each diagnoser state $y = \{(x_1, \gamma_1), \ldots, (x_k, \gamma_k)\}$ induces a diagnosis value*:*

$$\varphi(y) = \begin{cases} N & \text{if } \gamma_i = N \text{ for all } i \in \{1, \ldots, k\} \\ F & \text{if } \gamma_i = F \text{ for all } i \in \{1, \ldots, k\} \\ U & \text{otherwise} \end{cases}$$

For any observation $w \in \Sigma_o^*$, the diagnosis is $\varphi(w) = \varphi(\delta^*(y_0, w))$, where δ is the diagnoser's transition function.

Example 2.5 (Fault Diagnosis using Diagnoser)**.** *Consider a DFA $G = (X, \Sigma, \delta, x_0)$ with states $X = \{x_0, x_1, x_2\}$, alphabet $\Sigma = \{a, b, f\}$, and initial state x_0. The observable events are $\Sigma_o = \{a, b\}$ and the fault event is $f \in \Sigma_f$. The transition function contains $\delta(x_0, a) = x_1$, $\delta(x_0, f) = x_2$, $\delta(x_1, a) = x_1$, $\delta(x_2, a) = x_2$, and $\delta(x_2, b) = x_2$.*

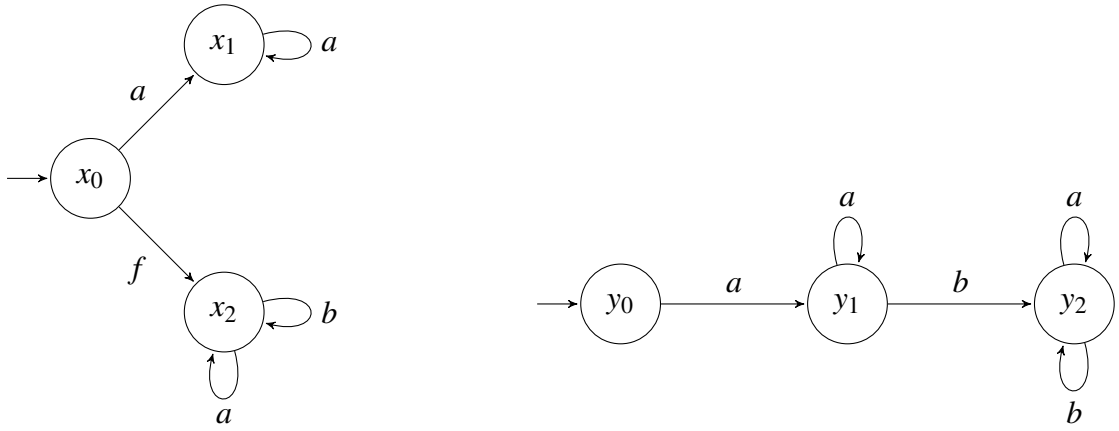


Figure 2.4: DFA G (left) and its diagnoser $\mathrm{Diag}(G)$ (right), where $y_0 = \{(x_0, N)\}$, $y_1 = \{(x_1, N), (x_2, F)\}$, and $y_2 = \{(x_2, F)\}$.

The diagnoser $\mathrm{Diag}(G)$ *has three states with diagnosis values: $\varphi(y_0) = N$ (no fault), $\varphi(y_1) = U$ (uncertain), and $\varphi(y_2) = F$ (fault). For observations: $\varphi(\varepsilon) = N$ (initial state,*

no fault occurred), $\varphi(a) = U$ *(ambiguous: could be in* x_1 *via* a *or in* x_2 *via unobservable* f *followed by* a*),* $\varphi(ab) = F$ *(definite fault: only state* x_2 *has transition* b*, which is reachable only through fault event* f*).*

This system is diagnosable*: after fault* f *occurs, observing event* b *definitively reveals the fault since* b *is only enabled in* x_2*, which is reachable only via* f*.*

Chapter 3

Switching Output Automata

This chapter establishes the mathematical foundations for modeling partially observable cyber-physical systems under timing constraints. We introduce the Switching Output Automaton (SOA) as the baseline formalism, which captures systems where internal discrete states are not directly observable, but generate piecewise-constant output signals that can be measured by external observers. Unlike classical discrete-event automata that operate over untimed event sequences, SOA is a *continuous-time discrete-event model*: state transitions and output switches occur at real-valued time instants, and the output observation signal $y_{\text{obs}}(t)$ is defined over continuous time $t \in \mathbb{R}_{\geq 0}$.

SOA extends classical automata in three key ways: (i) each state is associated with a set of possible output values rather than a single output; (ii) outputs can switch within a state, decoupling output changes from state transitions; and (iii) a *minimal dwell time* δ enforces that any change in the system—whether a discrete state transition or an output switch—must be separated from the previous change by at least δ time units. This dwell-time constraint reflects physical reality (stabilization time, sensor sampling intervals, actuator delays) and mathematically ensures Zeno-free behavior, guaranteeing only finitely many events occur in any bounded time interval.

From an observational perspective, the set-valued nature of the output mapping creates observational ambiguity: multiple internal states may produce the same output symbol, preventing direct state identification from observations. To enable state estimation and algorithmic analysis under partial observability, we construct an *evolution automaton*, a nondeterministic finite automaton (NFA) that abstracts the SOA's continuous-time behavior into discrete logical states tracking both global states and dwell-time satisfaction. We then apply subset construction, extending the NFA observer methods from Chapter 2, to obtain a deterministic observer that computes the set of logical states consistent with each output observation sequence. This observer enables online state estimation and supports

the opacity verification and fault diagnosis frameworks developed in Chapters 4 and 5.

The chapter is organized as follows. Section 3.1 presents the formal definition of SOA, develops the concepts of state runs, output runs, and state-output runs, introduces global states as pairs (x,y), and classifies transitions into three fundamental types. Section 3.1.3 defines the output observation signal $y_{\text{obs}}(t)$ and the system language $L_\delta(G)$ from the external observer's perspective. Section 3.2 constructs the evolution automaton by splitting each global state into waiting and ready phases to track dwell-time satisfaction. Section 3.3 presents the observer construction via subset construction and state simplification.

3.1 The Switching Output Automaton

3.1.1 Formal Definition

Definition 3.1 (Switching Output Automaton)**.** *A switching output automaton (SOA) is a six-tuple* $G = (X, Y, B, h, x_0, y_0)$*, where*

- X *is a finite set of states;*
- Y *is a finite* output alphabet*;*
- $B \subseteq \{(x,x') \in X \times X \mid x \neq x'\}$ *is a set of directed arcs (no self-loops);*
- $h : X \to 2^Y \setminus \{\emptyset\}$ *is the* output function *that associates each state with a non-empty set of possible output symbols;*
- $x_0 \in X$ *is the initial state;*
- $y_0 \in h(x_0)$ *is the initial output symbol.*

The output function $h(x) \subseteq Y$ defines the set of all possible output symbols that may be produced when the system resides in state x. Note that $h(x)$ is non-empty for all $x \in X$, ensuring that every state has at least one admissible output symbol. Consequently, the system always produces some output symbol—there is no "silent" or "empty" output. The output alphabet Y consists of concrete observable symbols; the empty word ε is not an element of Y. Unlike classical (untimed) automata, the *set-valued* output map h, together with the timing structure introduced below (output-switching instants and the minimal dwell-time constraint), enables SOA to model piecewise-constant, continuous-time outputs while the discrete state remains unchanged. In particular, h constrains the

admissible output values in each state, whereas the temporal evolution of $y(\cdot)$ is induced by the switching times and minimal dwell-time semantics.

For the structural properties of the automaton, an arc $b = (x, x') \in B$ represents a directed transition from state x to state x'. In this context, state x' is referred to as a *direct successor* of state x, while state x is called a *direct predecessor* of state x'. We denote by

$$\sigma(x) = \{x' \in X \mid (x, x') \in B\} \tag{3.1}$$

the set of all direct successors of state x. Similarly, we define the set of direct predecessors of state x' as

$$\sigma^{-1}(x') = \{x \in X \mid (x, x') \in B\}. \tag{3.2}$$

Example 3.1 (Power Monitoring System)**.** *Consider an SOA modeling a power monitoring system with three operational modes. Let* $G = (X, Y, B, h, x_0, y_0)$ *where:*

- $X = \{x_0, x_1, x_2\}$ *represents three states: idle* (x_0)*, active* (x_1)*, and critical* (x_2)*;*
- $Y = \{y_a, y_b, y_c\}$ *represents three distinct power consumption levels (e.g., low, medium, high);*
- $B = \{(x_0, x_1), (x_1, x_2), (x_2, x_0), (x_1, x_0)\}$ *defines the possible state transitions;*
- $h(x_0) = \{y_a\}$, $h(x_1) = \{y_a, y_b\}$, $h(x_2) = \{y_c\}$ *specify the output possibilities in each state;*
- *The initial state is* x_0*;*
- *The initial output is* $y_0 = y_a \in h(x_0)$.

In this example, the system starts in the idle state x_0 *with low power consumption* y_a*. While in state* x_1 *(active mode), the system can produce either output* y_a *(low power under light workload) or* y_b *(medium power under heavy workload), capturing the fact that power consumption may fluctuate between two levels as the workload varies during the active phase. State* x_2 *(critical mode) can only produce output* y_c *(high power), representing an overload or fault condition where the system operates at maximum power consumption. The arc structure shows that:*

- $\sigma(x_0) = \{x_1\}$*: from idle, the system can only transition to active;*
- $\sigma(x_1) = \{x_0, x_2\}$*: from active, the system can return to idle or escalate to critical;*
- $\sigma(x_2) = \{x_0\}$*: from critical, the system returns to idle.*

Figure 3.1 provides a graphical representation of this SOA, where each state is annotated with its possible output set.

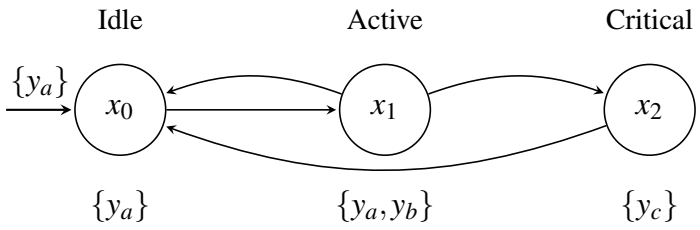


Figure 3.1: Graphical representation of the SOA from Example 3.1.

3.1.2 State-Output Runs and System Evolution

In classical automata theory, the behavior of a system is fully characterized by its state transitions *and the outputs they produce*. However, in the SOA framework, we must capture not only the discrete state evolution but also the continuous-time output switching within each state. This dual representation is formalized through the concept of state-output runs.

3.1.2.1 State Runs

The discrete evolution of states in an SOA can be described by means of a *state run*, which is a sequence

$$\xrightarrow{t_0} x^{(0)} \xrightarrow{t_1} x^{(1)} \xrightarrow{t_2} \cdots \xrightarrow{t_k} x^{(k)} \tag{3.3}$$

where:

- $x^{(i)} \in X$ for all $i = 0, \ldots, k$ are the visited states;
- $(x^{(i)}, x^{(i+1)}) \in B$ for all $i = 0, \ldots, k-1$ ensures that consecutive states are connected by valid arcs;
- $t_i \in \mathbb{R}_{\geq 0}$ with $t_0 = 0$ and $t_0 < t_1 < \cdots < t_k$ are the state transition times;
- $k \geq 0$ is the *length* of the state run (number of transitions).

This state run describes an evolution process where the automaton initially resides in state $x^{(0)}$ at time $t_0 = 0$, and subsequently transitions from state $x^{(i-1)}$ to state $x^{(i)}$ at time t_i for each $i = 1, \ldots, k$.

For a run to be valid in the SOA $G = (X, Y, B, h, x_0, y_0)$, we require that $x^{(0)} = x_0$, i.e., all runs must start from the initial state specified in the SOA definition.

Example 3.2 (State Run)**.** *Consider the SOA from Example 3.1. A possible state run is:*

$$\xrightarrow{t_0=0} x_0 \xrightarrow{t_1=3} x_1 \xrightarrow{t_2=7} x_2 \xrightarrow{t_3=10} x_0$$

This describes a cyclic path: idle → active → critical → idle, with state transitions occurring at times $t_0 = 0$, $t_1 = 3$, $t_2 = 7$, and $t_3 = 10$. The run has length $k = 3$ (three transitions).

3.1.2.2 Output Runs Within States

While the automaton remains in state $x^{(i)}$, its output may switch among the values in $h(x^{(i)})$. This intra-state output behavior is captured by an *output run* associated with state $x^{(i)}$:

$$\xrightarrow[\tau_0^{(i)}]{t_i} y_0^{(i)} \xrightarrow[\tau_1^{(i)}]{} y_1^{(i)} \xrightarrow[\tau_2^{(i)}]{} \cdots \xrightarrow[\tau_{m_i}^{(i)}]{} y_{m_i}^{(i)} \tag{3.4}$$

where:

- $y_j^{(i)} \in h(x^{(i)})$ for all $j = 0, \ldots, m_i$ are the output values produced while in state $x^{(i)}$;
- $y_{j-1}^{(i)} \neq y_j^{(i)}$ for all $j = 1, \ldots, m_i$ ensures that consecutive output values within the same state are distinct (this constraint does not apply across state transitions—the output may remain unchanged when transitioning between states);
- $\tau_j^{(i)} \in \mathbb{R}_{\geq 0}$ with $\tau_0^{(i)} = t_i < \tau_1^{(i)} < \cdots < \tau_{m_i}^{(i)}$ are the output switching times, and for $i = 0, \ldots, k-1$, we additionally require $\tau_{m_i}^{(i)} < t_{i+1}$ (the last output switch must occur before the next state transition);
- $m_i \geq 0$ is the *length* of the output run (number of output switches within state $x^{(i)}$).

When state $x^{(i)}$ is entered at time $\tau_0^{(i)} = t_i$, the output immediately takes the initial value $y_0^{(i)} \in h(x^{(i)})$. Subsequently, the output changes from $y_{j-1}^{(i)}$ to $y_j^{(i)}$ at time $\tau_j^{(i)}$ for each $j = 1, \ldots, m_i$. The output then remains at value $y_{m_i}^{(i)}$ until the next state transition occurs (or indefinitely if $i = k$).

Example 3.3 (Output Run Within a State)**.** *Continuing from Example 3.2, consider the output behavior while the system is in state x_1 (active mode) during the time interval $[3,7)$. Since $h(x_1) = \{y_a, y_b\}$, the output can switch between these two values. A possible output run is:*

$$\xrightarrow[\tau_0^{(1)}=3]{t_1=3} y_a \xrightarrow[\tau_1^{(1)}=5]{} y_b$$

This output run has length $m_1 = 1$ (one output switch). The system produces output y_a during $[3,5)$, then switches to y_b at time 5, and maintains y_b during $[5,7)$ until the next state transition occurs.

3.1.2.3 State-Output Runs and Global States

By combining state runs with their associated output runs, we obtain a complete description of the system's behavior through a *state-output run*:

$$\begin{aligned}
&\xrightarrow[\tau_0^{(0)}=0]{t_0=0} \begin{pmatrix} x^{(0)} \\ y_0^{(0)} \end{pmatrix} \xrightarrow[\tau_1^{(0)}]{} \begin{pmatrix} x^{(0)} \\ y_1^{(0)} \end{pmatrix} \xrightarrow[\tau_2^{(0)}]{} \cdots \xrightarrow[\tau_{m_0}^{(0)}]{} \begin{pmatrix} x^{(0)} \\ y_{m_0}^{(0)} \end{pmatrix} \xrightarrow[\tau_0^{(1)}]{t_1} \\
&\quad \begin{pmatrix} x^{(1)} \\ y_0^{(1)} \end{pmatrix} \xrightarrow[\tau_1^{(1)}]{} \cdots \xrightarrow[\tau_0^{(i)}]{t_i} \begin{pmatrix} x^{(i)} \\ y_0^{(i)} \end{pmatrix} \xrightarrow[\tau_1^{(i)}]{} \cdots \xrightarrow[\tau_{m_i}^{(i)}]{} \begin{pmatrix} x^{(i)} \\ y_{m_i}^{(i)} \end{pmatrix} \\
&\quad \xrightarrow[\tau_0^{(i+1)}]{t_{i+1}} \begin{pmatrix} x^{(i+1)} \\ y_0^{(i+1)} \end{pmatrix} \xrightarrow[\tau_1^{(i+1)}]{} \cdots \xrightarrow[\tau_{m_k}^{(k)}]{} \begin{pmatrix} x^{(k)} \\ y_{m_k}^{(k)} \end{pmatrix}
\end{aligned} \tag{3.5}$$

Each element in this run is a *global state* $q = (x, y) \in Q$, where $Q = \{(x, y) \mid x \in X, y \in h(x)\}$ is the *global state space*. A global state is thus an ordered pair consisting of:

1. A discrete state $x \in X$ representing the current mode;
2. An output value $y \in h(x) \subseteq Y$ representing the current observable signal.

The global state space Q captures all possible instantaneous configurations of the system. Since $Q \subseteq X \times Y$, the cardinality satisfies

$$n_Q = |Q| = \sum_{x \in X} |h(x)| \leq |X| \cdot |Y|,$$

with equality if and only if $h(x) = Y$ for all $x \in X$.

Example 3.4 (Complete State-Output Run)**.** *Combining Examples 3.2 and 3.3, the complete state-output run for the system is:*

$$\xrightarrow[\tau_0^{(0)}=0]{t_0=0} \begin{pmatrix} x_0 \\ y_a \end{pmatrix} \xrightarrow[\tau_0^{(1)}=3]{t_1=3} \begin{pmatrix} x_1 \\ y_a \end{pmatrix} \xrightarrow[\tau_1^{(1)}=5]{} \begin{pmatrix} x_1 \\ y_b \end{pmatrix} \xrightarrow[\tau_0^{(2)}=7]{t_2=7} \begin{pmatrix} x_2 \\ y_c \end{pmatrix} \xrightarrow[\tau_0^{(3)}=10]{t_3=10} \begin{pmatrix} x_0 \\ y_a \end{pmatrix}$$

The global state space for this SOA is $Q = \{(x_0, y_a), (x_1, y_a), (x_1, y_b), (x_2, y_c)\}$ *with cardinality* $|Q| = |h(x_0)| + |h(x_1)| + |h(x_2)| = 1 + 2 + 1 = 4$.

3.1.2.4 Classification of Transitions

An SOA can exhibit multiple state-output runs, each consisting of transitions that can be classified into three fundamental types based on how the discrete state and output evolve.

Type 1: State change with no output change. During the time interval $[\tau_{m_i}^{(i)}, t_{i+1})$, the system remains in state $x^{(i)}$ and produces output $y_{m_i}^{(i)}$. At time instant t_{i+1}, the system transitions to state $x^{(i+1)}$ while maintaining the same output value, i.e., $y_{m_i}^{(i)} = y_0^{(i+1)}$. This represents a mode change with no observable output change.

Type 2: Simultaneous state and output change. During the time interval $[\tau_{m_i}^{(i)}, t_{i+1})$, the system remains in state $x^{(i)}$ and produces output $y_{m_i}^{(i)}$. At time instant t_{i+1}, the system transitions to state $x^{(i+1)}$ and simultaneously generates a new output, which means that $y_{m_i}^{(i)} \neq y_0^{(i+1)}$. This represents a combined mode and output change.

Type 3: Output change with no state change. While the system remains in state $x^{(i)}$, the output undergoes m_i changes, represented by the sequence of output switches:

$$y_0^{(i)} \to y_1^{(i)} \to \cdots \to y_{m_i}^{(i)}.$$

Example 3.5 (Transition Type Classification)**.** *In the state-output run from Example 3.4, the transitions can be classified as:*

- ***Type 1*** *(at $t_1 = 3$): State changes from x_0 to x_1, but output remains y_a (since $y_{m_0}^{(0)} = y_a = y_0^{(1)}$).*
- ***Type 3*** *(at $\tau_1^{(1)} = 5$): Output changes from y_a to y_b while remaining in state x_1.*
- ***Type 2*** *(at $t_2 = 7$): State changes from x_1 to x_2 and output simultaneously changes from y_b to y_c (since $y_{m_1}^{(1)} = y_b \neq y_c = y_0^{(2)}$).*
- ***Type 2*** *(at $t_3 = 10$): State changes from x_2 to x_0 and output changes from y_c to y_a.*

3.1.2.5 Minimal Dwell Time Constraint and Zeno-Free Behavior

To ensure physically realizable system behavior, we impose a minimal temporal separation between consecutive transitions through the concept of *dwell time*. This constraint prevents *Zeno behavior*, where an infinite number of discrete transitions would occur within a finite time interval—a pathological phenomenon incompatible with physical systems.

Definition 3.2 (Minimal Dwell Time)**.** *A minimal dwell time $\delta \in \mathbb{R}_{>0}$ is a positive constant that specifies the minimal time duration that must elapse between any two consecutive transitions (of any type) in a state-output run.*

Formally, for a state-output run to satisfy the minimal dwell time constraint δ, the following conditions must hold:

1. **Between consecutive output switches within the same state:**

$$\tau_{j+1}^{(i)} - \tau_j^{(i)} \geq \delta, \quad \forall i = 0, \ldots, k, \quad \forall j = 0, \ldots, m_i - 1 \tag{3.6}$$

   Note that when $m_i = 0$ (i.e., no output switches occur within state $x^{(i)}$), the constraint (3.6) becomes vacuous as the index set for j is empty, and thus is automatically satisfied.

2. **Between the last output switch and the next state transition:**

$$\tau_0^{(i+1)} - \tau_{m_i}^{(i)} = t_{i+1} - \tau_{m_i}^{(i)} \geq \delta, \quad \forall i = 0, \ldots, k-1 \tag{3.7}$$

Equivalently, let

$$\mathscr{T} := \{t_i : i = 0, \ldots, k\} \cup \bigcup_{i=0}^{k} \{\tau_j^{(i)} : j = 1, \ldots, m_i\}.$$

The minimal dwell-time constraint requires that any two consecutive instants $u < v$ in $\mathscr{T}$ satisfy $v - u \geq \delta$. Equality is allowed: two events may be separated by exactly δ time units.

By enforcing $\delta > 0$, the minimal dwell time constraint guarantees that:

- Any finite time interval contains only a finite number of transitions;
- The system evolution is well-defined and amenable to formal verification techniques;
- The model accurately reflects real physical systems, which inherently require non-zero time for state changes and output switching.

In practical applications, the dwell time δ is typically determined by the physical characteristics of the system, such as sensor sampling rates, actuator response times, or minimal process durations. This constraint represents a deliberate modeling assumption: by requiring $\delta > 0$, the SOA framework excludes behaviors with arbitrarily fast switching or Zeno-like accumulation of events. If a target application truly requires modeling arbitrarily short inter-event times, a different model class would be necessary. In practice, δ is chosen to match the system's temporal resolution, ensuring that SOA captures the

physically meaningful behaviors at the granularity relevant for observation and verification.

Example 3.6 (Minimal Dwell Time Verification)**.** *For the state-output run in Example 3.4, let us verify that it satisfies the minimal dwell time constraint with $\delta = 1$ time unit. All consecutive transitions must satisfy:*

- $t_1 - t_0 = 3 - 0 = 3 \geq 1$ ✓
- $\tau_1^{(1)} - \tau_0^{(1)} = 5 - 3 = 2 \geq 1$ ✓*(output switch within state x_1)*
- $t_2 - \tau_1^{(1)} = 7 - 5 = 2 \geq 1$ ✓*(from last output switch to state transition)*
- $t_3 - t_2 = 10 - 7 = 3 \geq 1$ ✓

3.1.3 Observations and System Behavior

In many practical scenarios involving cyber-physical systems, an external entity (such as a monitoring system, a diagnoser, or even an adversary) does not have direct access to the internal discrete state of the system. Instead, the only available information is the time-indexed output signal produced by the system. This section formalizes the notion of observations and introduces the concept of system behavior as perceived by an external observer.

3.1.3.1 Output Observation Signal

Notation (output symbols vs. output signal). Throughout the thesis, we use $y \in Y$ to denote an *output symbol* (e.g., the output component of a global state $q = (x, y)$). The *time-indexed output signal observed along a run* is denoted by $y_{\text{obs}} : \mathbb{R}_{\geq 0} \to Y$. Hence, $y_{\text{obs}}(t)$ is the output symbol produced at time t, while y without an argument denotes a symbol in the finite alphabet Y.

Definition 3.3 (Output Observation Signal)**.** *Given an SOA $G = (X, Y, B, h, x_0, y_0)$ executing a state-output run, the* output observation signal *is a mapping $y_{\text{obs}} : T \to Y$, where $T = \mathbb{R}_{\geq 0}$ represents continuous time, such that $y_{\text{obs}}(t)$ denotes the output symbol produced by the system at time instant $t \in T$.*

The time axis is partitioned at the output switching times. There exists a strictly increasing sequence $(s_i)_{i \in I}$ with $s_0 = 0$ such that $y_{\text{obs}}(t) = y_i$ for all $t \in [s_i, s_{i+1})$ and adjacent values are distinct. If the number of switches is finite (i.e., $I = \{0, 1, \ldots, n\}$ for some n), we

set $s_{n+1} = \infty$, so the last interval is $[s_n, \infty)$. Otherwise $I = \mathbb{N}$ and, by the minimal dwell-time constraint (Definition 3.2) with $\delta > 0$, one has $s_i \uparrow \infty$, yielding a countably infinite partition of $[0, \infty)$ with no terminal interval.

Proposition 3.1 (Properties of the observed output signal under minimal dwell time)**.** *Assume the underlying state-output run satisfies the minimal dwell-time constraint with $\delta > 0$. Then the observed output signal $y_{\text{obs}}(t)$ satisfies the following properties:*

1. ***Piecewise constant**: $y_{\text{obs}}(t)$ is constant within each time interval $[s_i, s_{i+1})$ and may change only at the partition points $\{s_i\}$;*
2. ***Minimal dwell time constraint**: Each output value persists for a duration of at least $\delta > 0$, i.e., if $y_{\text{obs}}(t) = y'$ for all $t \in [s_i, s_{i+1})$, then $s_{i+1} - s_i \geq \delta$;*
3. ***Distinct consecutive values**: The output values in adjacent time intervals are different, i.e., if $y_{\text{obs}}(t) = y'$ for $t \in [s_i, s_{i+1})$ and $y_{\text{obs}}(t) = y''$ for $t \in [s_{i+1}, s_{i+2})$, then $y' \neq y''$.*

Proof. The properties follow directly from the definitions of state-output runs and the minimal dwell-time constraint. □

3.1.3.2 Observation Sequences and System Language

To represent observations in a compact form, we introduce the notion of *timed output symbols*.

Definition 3.4 (Timed Output Symbol)**.** *A* timed output symbol *is a pair (y, τ) where $y \in Y$ is an output value and $\tau \in \mathbb{R}_{\geq \delta}$ is the duration for which the output remains at value y.*

An observation sequence is then a finite string of timed output symbols, representing the complete output trace observed over a finite time horizon.

Definition 3.5 (Language of an SOA under minimal dwell time δ)**.** *Fix $\delta > 0$. The* language *$L_\delta(G)$ is the set of all observation sequences $\omega \in (Y \times \mathbb{R}_{\geq \delta})^+$ for which there exist a state-output run **that satisfies the minimal dwell-time constraint with** δ, and a partition $0 = s_0 < s_1 < \cdots < s_{n+1}$ such that $\omega = (y_0, \tau_0) \cdots (y_n, \tau_n)$, $y_{\text{obs}}(t) = y_i$ for $t \in [s_i, s_{i+1})$, $\tau_i = s_{i+1} - s_i \geq \delta$, and $y_i \neq y_{i+1}$ for $i = 0, \ldots, n-1$. Here $(Y \times \mathbb{R}_{\geq \delta})^+$ denotes the set of all finite-length non-empty sequences of timed output symbols (where $Y \times \mathbb{R}_{\geq \delta}$ is the Cartesian product, i.e., the set of all pairs (y, τ) with $y \in Y$ and $\tau \geq \delta$).*

Remark 3.1 (Finite vs. Infinite Observations)**.** *The language $L_\delta(G)$ consists of finite observation sequences, reflecting practical scenarios where systems are observed over*

bounded time horizons. While the SOA may continue executing indefinitely (in the finite-switch case, the partition ends with $[s_n, \infty)$), we only consider finite prefixes of the system's behavior. If one were to consider infinite-horizon observations, the corresponding language would consist of ω-sequences (infinite sequences) of timed output symbols, with each symbol still satisfying the minimal dwell time constraint $\tau_i \geq \delta$. This ensures the absence of Zeno behavior even in the infinite case.

Each observation sequence $\omega \in L_\delta(G)$ encodes both the qualitative evolution (the sequence of output symbols) and the quantitative timing information (the duration of each output). This dual representation is essential for capturing the timed behavior of cyber-physical systems.

Example 3.7 (Observation Sequence)**.** *Consider the state-output run from Example 3.4. The corresponding output observation signal is:*

$$y_{\text{obs}}(t) = \begin{cases} y_a & \text{if } t \in [0,5) \\ y_b & \text{if } t \in [5,7) \\ y_c & \text{if } t \in [7,10) \\ y_a & \text{if } t \in [10,\infty) \end{cases}$$

Note that while the system continues executing beyond time 10 (remaining in state x_0 with output y_a), we only observe the system over the finite interval $[0,10]$.

The observation sequence over this time interval is:

$$\omega = (y_a,5)(y_b,2)(y_c,3)$$

This sequence $\omega \in L_\delta(G)$ (with $\delta = 1$) satisfies all requirements:

- *Consecutive outputs are distinct: $y_a \neq y_b$ and $y_b \neq y_c$.*
- *All durations satisfy the minimal dwell time: $5 \geq 1$, $2 \geq 1$, $3 \geq 1$.*

Note that from this observation alone, an external observer knows:

- *The output was y_a for 5 time units (covering both the initial stay in x_0 and part of the stay in x_1);*
- *Then switched to y_b for 2 time units (remaining time in x_1);*
- *Then switched to y_c for 3 time units (entire stay in x_2).*

However, the observer cannot distinguish between the two phases within the first 5 time units where the system produced y_a while being in different states (x_0 vs. x_1).

Figure 3.2 illustrates the output observation signal $y_{\text{obs}}(t)$ as a piecewise constant function over time.

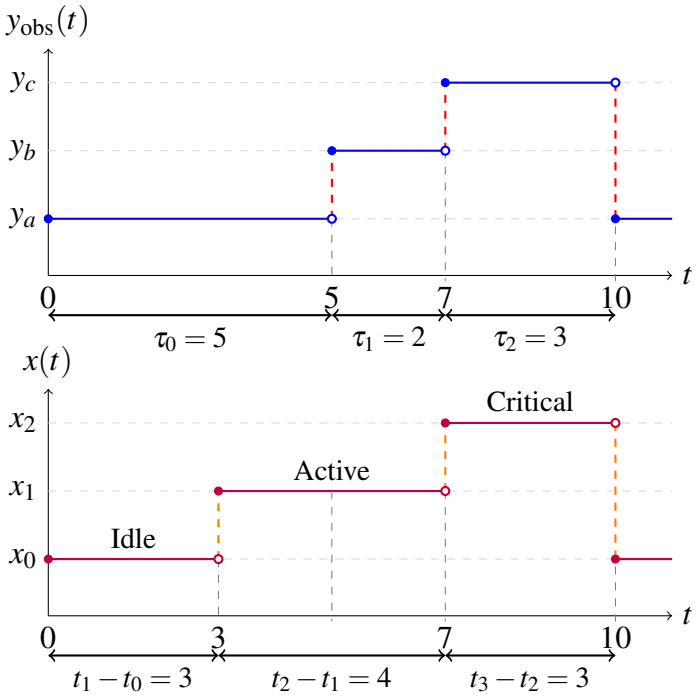


Figure 3.2: Output observation signal $y_{\text{obs}}(t)$ (top) and state function $x(t)$ (bottom) for Example 3.4.

3.2 Evolution Automaton

To enable algorithmic verification and analysis of SOAs, we construct a finite-state abstraction called the *evolution automaton*. This nondeterministic automaton G_e represents the evolution of an SOA by tracking both the current global state (x,y) and whether the minimal dwell time constraint has been satisfied.

The key challenge is that the SOA's dwell time constraint requires any change in the system (whether a state transition or output switch) to be separated from the previous change by at least δ time units. To capture this timing constraint in a finite automaton, each global state $q = (x,y) \in X \times Y$ is split into two phases based on whether the dwell time has expired. Specifically, we introduce two types of discrete states for each $x \in X$: a *waiting discrete state* x' indicating that the minimal dwell time has not yet been satisfied since the last change, and a *ready discrete state* x'' indicating that the minimal dwell time has been satisfied. We define $X' = \{x' \mid x \in X\}$ as the set of all waiting discrete states

and $X'' = \{x'' \mid x \in X\}$ as the set of all ready discrete states. Let $X_e = X' \cup X''$ denote this extended discrete state space.

We construct the states of the evolution automaton by pairing each waiting or ready discrete state with an output. These states, called *logical states*, have the form $q_e = (x_e, y)$ where $x_e \in X_e$ and $y \in Y$. Thus, each global state (x, y) of the original SOA is represented by two logical states: (x', y) when the dwell time constraint is not yet satisfied, and (x'', y) when it has been satisfied. We emphasize that the waiting/ready distinction is an internal modeling mechanism for verification purposes; an external observer does not directly observe whether the system is in a waiting or ready state. The observer only measures the output symbol and its duration—the δ-transitions (waiting $\to$ ready) are unobservable. The observer can infer that at least δ time has passed when observing a duration $\geq \delta$, but cannot distinguish between (x', y) and (x'', y) from a single observation.

Definition 3.6 (Evolution Automaton)**.** *Given an SOA $G = (X, Y, B, h, x_0, y_0)$ and a minimal dwell time $\delta > 0$, the* evolution automaton *of G is a nondeterministic finite automaton $G_e = (Q_e, \Sigma_e, \Delta, q_{e,0})$ where*

- *$Q_e \subseteq X_e \times Y$ is the finite set of logical states;*
- *$\Sigma_e = Y \cup \{\delta\}$ is the event alphabet, where Y are observable output symbols and δ denotes the expiration of the minimal dwell time;*
- *$\Delta \subseteq Q_e \times (\Sigma_e \cup \{\varepsilon\}) \times Q_e$ is the transition relation, where ε denotes an unobservable transition;*
- *$q_{e,0} = (x_0', y_0)$ is the initial logical state.*

The construction of Q_e and Δ is given in Algorithm 1.

Remark 3.2 (Clarifications on Evolution Automaton Structure)**.** *Several aspects of the evolution automaton definition merit clarification:*

1. Richer structure than SOA. *The evolution automaton G_e captures more information than the original SOA G: its transition relation Δ explicitly represents both discrete state transitions and within-state output switches, as well as the waiting/ready phase distinction.*
2. No marked states. *Unlike standard NFA definitions, G_e has no marked (final) states because we use G_e for reachability-based verification (opacity, diagnosis) rather than language acceptance.*

3. Unobservable transitions. *A transition is unobservable (labeled ε) when the discrete state changes but the output remains the same—these correspond to Type 1 transitions defined in Section 4.3.2.4. An external observer cannot distinguish such transitions from continued residence in the previous global state.*

4. Infinite dwell. *If the system remains in a ready state (x'', y) indefinitely without further transitions, the evolution automaton models this via self-loops: repeated δ-transitions on (x'', y) represent arbitrarily long dwell times. No information is lost; the automaton simply stays in (x'', y).*

Remark 3.3 (Relationship to untimed abstractions and timed/hybrid models)**.** *Although SOA is a continuous-time discrete-event model, it admits a systematic finite-state representation tailored to observer-based verification. Conceptually, SOA can be viewed as a restricted timed-style model in which a single clock records the time since the most recent change (either a discrete state transition or an output switch); any change is enabled only after at least δ time units have elapsed, and the clock is reset to zero after the change. The set-valued output mapping $h(x)$ restricts admissible output values while the system resides in state x, and output switching within a state corresponds to changes of the output value without changing the discrete state.*

For algorithmic verification, rather than working with dense-time clock semantics, we construct the evolution automaton *G_e (Definition 3.6), an NFA that replaces the continuous clock with a two-phase waiting/ready abstraction. Each global state (x, y) is split into (x', y) (dwell time not yet satisfied) and (x'', y) (dwell time satisfied), and a special symbol δ represents the expiration of one minimal dwell-time period. This construction preserves exactly the enabling discipline induced by the minimal dwell-time constraint and provides a finite structure on which subset-based observers, opacity verification, and fault diagnosis procedures can be carried out in later chapters.*

The key insight is that SOA's minimal dwell-time constraint $\delta > 0$ bounds the rate of discrete transitions, enabling a finite discretization for verification purposes. This contrasts with general timed automata, where opacity verification under dense-time semantics is generally undecidable, and decidability requires strong restrictions (e.g., real-time automata with 2-EXPTIME complexity, discrete-time semantics, or constrained observation models). By designing SOA as a more structured subclass, we obtain decidable and implementable verification while retaining sufficient expressiveness for modeling partially observable cyber-physical systems.

The evolution automaton represents the evolution of an SOA through transitions between logical states. From a waiting discrete state $x' \in X'$, the system must wait until

the minimal dwell time δ has elapsed before any further action can occur. This waiting period is represented by a δ-labeled transition from (x', y) to (x'', y), which transitions the discrete state from waiting to ready while maintaining the same output.

Once in a ready discrete state $x'' \in X''$, the three types of transitions defined in Section 3.1.2 are represented in the evolution automaton as follows:

Type 1 (State change without output change): When the discrete state changes from x to $\bar{x} \in \sigma(x)$ but the output remains $y \in h(x) \cap h(\bar{x})$, an external observer cannot detect the state transition. This is represented by an ε-labeled transition from (x'', y) to $(\bar{x}', y)$ in the evolution automaton.

Type 2 (Simultaneous state and output change): When both the discrete state and output change simultaneously (from x to $\bar{x} \in \sigma(x)$ and from y to $\bar{y} \in h(\bar{x}) \setminus \{y\}$), the observer detects the output change $\bar{y}$ but cannot necessarily determine whether a state transition occurred. This is represented by a $\bar{y}$-labeled transition from (x'', y) to $(\bar{x}', \bar{y})$.

Type 3 (Output change without state change): When the output changes from y to $\bar{y} \in h(x) \setminus \{y\}$ while the discrete state remains x, the observer detects the output change. This is represented by a $\bar{y}$-labeled transition from (x'', y) to $(x', \bar{y})$.

Note that in all cases where the output changes (Types 2 and 3), the observer can detect that *some* transition occurred, but the ambiguity in output-to-state mapping (due to the set-valued output function h) prevents the observer from uniquely determining the current discrete state.

Algorithm 1 constructs the evolution automaton through a reachability analysis that explores all logical states accessible from the initial state $q_{e,0}$. For notational convenience, given an extended discrete state $x_e \in X_e$ of the form x' or x'', we denote by $x \in X$ the corresponding base discrete state (i.e., if $x_e = x'$ or $x_e = x''$, then the base state is x).

The algorithm performs a reachability-based exploration starting from the initial logical state $q_{e,0} = (x'_0, y_0)$. In each iteration, a logical state q_e is selected from $Q_{e,new}$ and processed: all reachable successor states and their corresponding transitions are added to the automaton according to the transition semantics described above. The state q_e is then moved from $Q_{e,new}$ to Q_e (the set of explored states), ensuring that each state is processed exactly once. The algorithm terminates when $Q_{e,new}$ becomes empty, guaranteeing that Q_e contains exactly the set of logical states reachable from $q_{e,0}$.

Algorithm 1 Constructing the evolution automaton

Require: SOA $G = (X, Y, B, h, x_0, y_0)$
Ensure: Evolution automaton $G_e = (Q_e, \Sigma_e, \Delta, q_{e,0})$

Set $X_e = X' \cup X'' = \{x' | x \in X\} \cup \{x'' | x \in X\}$
Set $\Sigma_e = Y \cup \{\delta\}$
Set $q_{e,0} = (x'_0, y_0)$, $Q_{e,new} = \{q_{e,0}\}$, $Q_e = \emptyset$
Set $\Delta = \emptyset$
while $Q_{e,new} \neq \emptyset$ **do**
    Select a logical state $q_e = (x_e, y) \in Q_{e,new}$
    if $x_e = x' \in X'$ **then** ▷ Waiting state
        $q'_e = (x'', y)$, $\Delta = \Delta \cup \{(q_e, \delta, q'_e)\}$ ▷ Transition to ready state
        if $q'_e \notin Q_{e,new} \cup Q_e$ **then**
            $Q_{e,new} = Q_{e,new} \cup \{q'_e\}$
        end if
    else if $x_e = x'' \in X''$ **then** ▷ Ready state
        $\Delta = \Delta \cup \{(q_e, \delta, q_e)\}$ ▷ Self-loop for time progression
        for each $\bar{x} \in \sigma(x)$ **do**
            if $y \in h(\bar{x})$ **then** ▷ Type 1: state change, same output
                $q'_e = (\bar{x}', y)$, $\Delta = \Delta \cup \{(q_e, \varepsilon, q'_e)\}$
                if $q'_e \notin Q_{e,new} \cup Q_e$ **then**
                    $Q_{e,new} = Q_{e,new} \cup \{q'_e\}$
                end if
            end if
            for each $\bar{y} \in h(\bar{x}) \setminus \{y\}$ **do** ▷ Type 2: state and output change
                $q'_e = (\bar{x}', \bar{y})$, $\Delta = \Delta \cup \{(q_e, \bar{y}, q'_e)\}$
                if $q'_e \notin Q_{e,new} \cup Q_e$ **then**
                    $Q_{e,new} = Q_{e,new} \cup \{q'_e\}$
                end if
            end for
        end for
        for each $\bar{y} \in h(x) \setminus \{y\}$ **do** ▷ Type 3: output change, same state
            $q'_e = (x', \bar{y})$, $\Delta = \Delta \cup \{(q_e, \bar{y}, q'_e)\}$
            if $q'_e \notin Q_{e,new} \cup Q_e$ **then**
                $Q_{e,new} = Q_{e,new} \cup \{q'_e\}$
            end if
        end for
    end if
    $Q_{e,new} = Q_{e,new} \setminus \{q_e\}$, $Q_e = Q_e \cup \{q_e\}$ ▷ Mark state as explored
end while

Example 3.8 (Evolution Automaton Construction)**.** *Consider the SOA from Example 3.1 with discrete states* $X = \{x_0, x_1, x_2\}$*, outputs* $Y = \{y_a, y_b, y_c\}$*, and arcs*

$$B = \{(x_0, x_1), (x_1, x_2), (x_2, x_0), (x_1, x_0)\}.$$

The output function is given by $h(x_0) = \{y_a\}$, $h(x_1) = \{y_a, y_b\}$, *and* $h(x_2) = \{y_c\}$.

Applying Algorithm 1, the evolution automaton G_e *is constructed starting from the initial logical state* $q_{e,0} = (x'_0, y_a)$.

The resulting evolution automaton contains eight reachable logical states:

$$Q_e = \{(x'_0, y_a), (x''_0, y_a), (x'_1, y_a), (x'_1, y_b), \\ (x''_1, y_a), (x''_1, y_b), (x'_2, y_c), (x''_2, y_c)\}$$

The transition relation Δ *includes three types of transitions:*

- ***δ-transitions*** *(dwell time expiration): Each waiting state transitions to its corresponding ready state after* δ *time has elapsed, e.g.,* $(x'_0, y_a) \xrightarrow{\delta} (x''_0, y_a)$.
- ***ε-transitions*** *(Type 1): State changes without output changes are unobservable, e.g.,* $(x''_0, y_a) \xrightarrow{\varepsilon} (x'_1, y_a)$ *represents the transition from* x_0 *to* x_1 *while maintaining output* y_a.
- ***Output-labeled transitions*** *(Types 2 and 3): Output changes are observable, e.g.,* $(x''_1, y_a) \xrightarrow{y_b} (x'_1, y_b)$ *represents an output change within state* x_1, *while* $(x''_1, y_b) \xrightarrow{y_c} (x'_2, y_c)$ *represents a simultaneous state and output change.*

Figure 3.3 shows the complete evolution automaton. Note that state-changing transitions (Types 1 and 2) and output-switching transitions (Type 3) reset the discrete state component to its waiting version (primed), re-enforcing the minimal dwell time constraint. In contrast, δ*-transitions from ready states to themselves (self-loops) maintain the ready status, allowing time to progress indefinitely while the system remains in the same global state.*

3.3 Observer

The evolution automaton G_e constructed in the previous section is a nondeterministic structure that describes all possible runs of an SOA G. While this representation captures the complete behavior of the system, its nondeterministic nature prevents direct use for online state estimation, which requires deterministic tracking of the set of states consistent with observations.

Given a sequence of observations $\omega \in (Y, \mathbb{R}_{\geq \delta})^+$, we denote by $C(\omega)$ the set of states in which the system can be after observing ω. The goal of this section is to construct an *observer*, a deterministic automaton that allows one to compute the set $C(\omega)$ for any

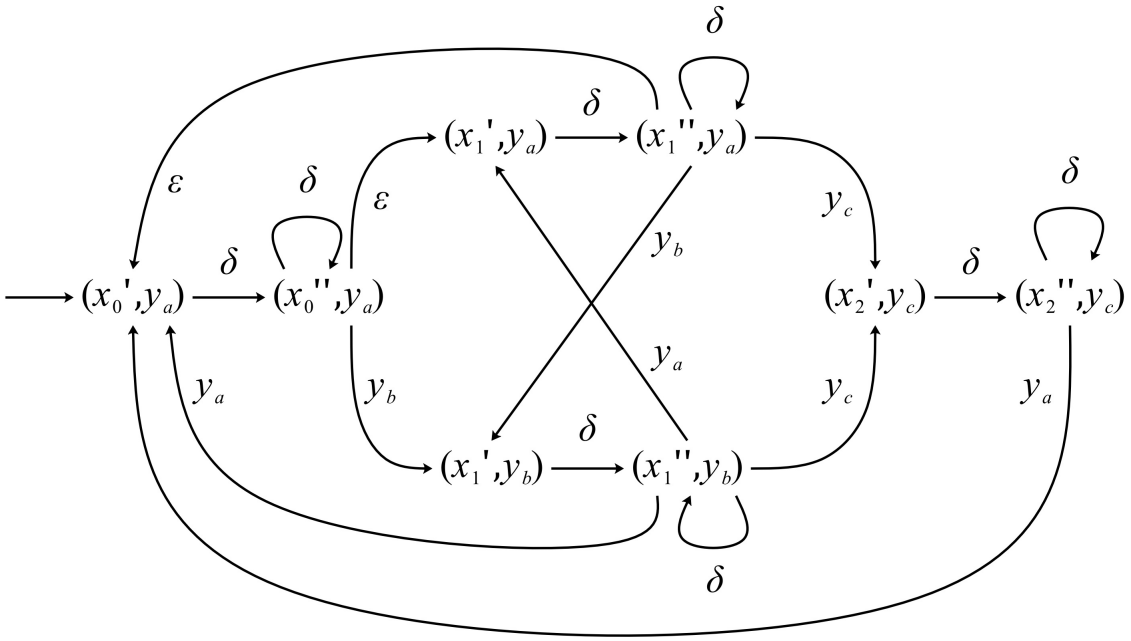


Figure 3.3: Evolution automaton for the power monitoring SOA from Example 3.1.

observed sequence ω without online computation. In this way, $C(\omega)$ can be computed simply by following a path in the observer.

An equivalent deterministic automaton can be obtained from the evolution automaton G_e using the standard subset construction. However, before presenting the observer construction algorithm, we need to introduce a state simplification procedure that exploits the structure of the evolution automaton.

3.3.1 State Simplification

In the standard subset construction for determinizing a nondeterministic automaton, each state of the deterministic automaton corresponds to a subset of states of the nondeterministic automaton. In our case, a naive application of this construction would yield observer states that are subsets of $Q_e \subseteq X_e \times Y$, the state set of the evolution automaton.

However, we can exploit the structure of the evolution automaton to obtain a more compact representation. Specifically, if an observer state contains both the waiting state (x',y) and the ready state (x'',y) for the same discrete state $x \in X$ and output $y \in Y$, we can simplify it by keeping only the ready state (x'',y). This is justified because: (i) from a waiting state (x',y), the only enabled transition is the δ-transition to the corresponding ready state (x'',y)—state transitions and output switches are blocked until the dwell time expires; (ii) the ready state (x'',y) enables all subsequent observable transitions (state changes and output switches). Therefore, if both (x',y) and (x'',y) are in an observer

state, the waiting state is redundant: any future behavior reachable from (x', y) must first pass through (x'', y). We can safely remove x' when x'' is also in the observer state.

This simplification allows us to represent observer states as elements of $Z \subseteq 2^{X_e} \times Y$ rather than 2^{Q_e}, where each observer state $z = (X_z, y)$ consists of a subset $X_z \subseteq X_e$ of extended discrete states and a single output symbol $y \in Y$.

We formalize this simplification using a function $\zeta : 2^{Q_e} \to 2^{X_e} \times Y$ defined as follows. For a subset $Q_{sub} \subseteq Q_e$ of evolution automaton states that all have the same output label y, we first extract the extended discrete states:

$$X_s(Q_{sub}) = \{x_e \in X_e \mid (x_e, y) \in Q_{sub}\}$$

Then we remove all waiting states $x' \in X'$ for which the corresponding ready state $x'' \in X''$ is also present:

$$X'_s(Q_{sub}) = X_s(Q_{sub}) \setminus \{x' \in X' \mid x'' \in X_s(Q_{sub}) \text{ for the same } x \in X\}$$

Finally, the simplification function is defined as:

$$\zeta(Q_{sub}) = (X'_s(Q_{sub}), y)$$

Algorithm 2 describes how to compute $\zeta(Q_{sub})$.

Algorithm 2 Computing $\zeta(Q_{sub})$

Require: $Q_{sub} \subseteq Q_e$, set of evolution automaton states with same output y
Ensure: $\zeta(Q_{sub}) \in 2^{X_e} \times Y$, simplified state
1: Set $X_s(Q_{sub}) = \{x_e \in X_e \mid (x_e, y) \in Q_{sub}\}$
2: **for all** $x_e, \bar{x}_e \in X_s(Q_{sub})$ **do**
3: **if** $x_e = x' \in X'$ and $\bar{x}_e = \bar{x}'' \in X''$ and $x = \bar{x}$ **then**
4: $X_s(Q_{sub}) = X_s(Q_{sub}) \setminus \{x_e\}$
5: **end if**
6: **end for**
7: **return** $\zeta(Q_{sub}) = (X_s(Q_{sub}), y)$

3.3.2 Observer Construction

We now formally define the observer as a deterministic finite automaton.

Definition 3.7 (Observer)**.** *Given an SOA $G = (X, Y, B, h, x_0, y_0)$ and its evolution automaton $G_e = (Q_e, \Sigma_e, \Delta, q_{e,0})$, the* observer *of G is a deterministic finite automaton $G_{obs} = (Z, \Sigma_e, \delta_o, z_0)$ where*

- $Z \subseteq 2^{X_e} \times Y$ *is the finite set of observer states;*
- $\Sigma_e = Y \cup \{\delta\}$ *is the event alphabet (same as for* G_e*);*
- $\delta_o : Z \times \Sigma_e \to Z$ *is a partial transition function;*
- $z_0 = (\{x'_0\}, y_0)$ *is the initial observer state.*

Each observer state $z = (X_z, y) \in Z$ is a pair where $X_z \subseteq X_e$ is a set of extended discrete states and $y \in Y$ is the current output. The observer tracks the set of evolution automaton states that are consistent with the observed sequence of outputs and time intervals.

Before presenting the construction algorithm, we introduce some useful notation for computing successor states in the evolution automaton:

- For each state $q \in Q_e$ of G_e, we define $D_\varepsilon(q)$ as the set of all states reachable from q by executing zero or more ε-transitions. Formally, $D_\varepsilon(q) = \{\bar{q} \in Q_e \mid$ there exists a path from q to $\bar{q}$ using only ε-transitions$\}$. Note that by definition $q \in D_\varepsilon(q)$.
- For each state $q \in Q_e$ and each symbol $y \in \Sigma_e$, we define $D_y(q) = \{\bar{q} \in Q_e \mid (q, y, \bar{q}) \in \Delta\}$ as the set containing all states reachable from q by executing exactly one observable y-transition.
- For each set of states $Q_{sub} \subseteq Q_e$ and each symbol $y \in \Sigma_e$, we define:

$$\alpha(Q_{sub}, y) = \bigcup_{q \in Q_{sub}} D_y(q)$$
$$\beta(Q_{sub}, y) = \bigcup_{q \in \alpha(Q_{sub}, y)} D_\varepsilon(q)$$

  as the set of states reachable from Q_{sub} by first executing one observable y-transition, then following zero or more ε-transitions.

Algorithm 3 describes the construction of the observer using the standard subset construction combined with the state simplification function ζ.

The algorithm performs a breadth-first exploration of the observer state space. Starting from the initial state z'_0, which contains all evolution automaton states reachable from $q_{e,0}$ via ε-transitions, the algorithm computes successor states for each event $y \in \Sigma_e$.

For each event $y \in \Sigma_e$, the algorithm computes the set $\alpha(z', y)$ of evolution automaton states reachable by taking one y-labeled transition from states in z', then computes $\beta(z', y)$ by following all ε-transitions from states in $\alpha(z', y)$.

Algorithm 3 Constructing the observer

Require: $G_e = (Q_e, \Sigma_e, \Delta, q_{e,0})$, the evolution automaton
Ensure: $G_{obs} = (Z, \Sigma_e, \delta_o, z_0)$, the observer
1: Set $z'_0 = D_\varepsilon(q_{e,0})$, $Z' = \{z'_0\}$, mark z'_0 as unprocessed
2: Set $z_0 = \zeta(z'_0)$, $Z = \{z_0\}$
3: **while** there exist unprocessed states in Z' **do**
4: Select an unprocessed state $z' \in Z'$
5: **for all** $y \in \Sigma_e$ **do**
6: $\alpha(z', y) = \bigcup_{q \in z'} D_y(q)$
7: $\beta(z', y) = \bigcup_{q \in \alpha(z', y)} D_\varepsilon(q)$
8: $\bar{z}' = \beta(z', y)$
9: **if** $\bar{z}' \neq \emptyset$ and $\bar{z}' \notin Z'$ **then**
10: $Z' = Z' \cup \{\bar{z}'\}$, mark $\bar{z}'$ as unprocessed
11: **end if**
12: $z = \zeta(z')$
13: $\bar{z} = \zeta(\bar{z}')$, $Z = Z \cup \{\bar{z}\}$
14: **if** $\bar{z} \neq \emptyset$ and $z \neq \bar{z}$ **then**
15: $\delta_o(z, y) = \bar{z}$
16: **end if**
17: **end for**
18: Mark z' as processed
19: **end while**

For each computed successor state $\bar{z}'$, the algorithm applies the simplification function ζ to obtain the corresponding simplified observer state $\bar{z}$, and adds the transition $(z, y, \bar{z})$ to the observer.

Example 3.9 (Observer Construction)**.** *Consider the SOA from Example 3.1 and its evolution automaton from Example 3.8. We construct the observer G_{obs} by applying Algorithm 3.*

Starting from the initial state $z'_0 = D_\varepsilon(q_{e,0}) = D_\varepsilon((x'_0, y_a)) = \{(x'_0, y_a)\}$, the algorithm explores all reachable observer states. After simplification using ζ, the initial observer state is $z_0 = \zeta(z'_0) = (\{x'_0\}, y_a)$.

The complete observer contains the following reachable states (after simplification):

$$\begin{aligned} Z = \{&(\{x'_0\}, y_a), (\{x''_0\}, y_a), (\{x''_0, x'_1\}, y_a), \\ &(\{x''_1\}, y_a), (\{x'_1\}, y_b), (\{x''_1\}, y_b), \\ &(\{x'_2\}, y_c), (\{x''_1, x'_2\}, y_c), (\{x''_2\}, y_c)\} \end{aligned}$$

For example, when the evolution automaton state set $\{(x''_0, y_a), (x'_1, y_a)\}$ is encountered, it is simplified to $(\{x''_0, x'_1\}, y_a)$ without removing any extended discrete state since

they correspond to different base states (x_0 and x_1).

Figure 3.4 shows the complete observer. The deterministic structure allows efficient state estimation: given any observation sequence ω, we simply follow the corresponding path in the observer to determine $C(\omega)$.

For instance, consider the observation sequence $\omega = (y_a, \delta)(y_a, \delta)(y_b, \delta)$, which means:

- *Output y_a for time δ*
- *Continue with output y_a for another time δ*
- *Change to output y_b for time δ*

Following this sequence in the observer:

$$(\{x_0'\}, y_a) \xrightarrow{\delta} (\{x_0''\}, y_a) \xrightarrow{\delta} (\{x_0'', x_1'\}, y_a) \xrightarrow{y_b} (\{x_1'\}, y_b)$$

Note that the first two δ-transitions occur while the output remains y_a. The observer infers these transitions not from output changes, but from the observed duration*: when the observer measures that output y_a has persisted for at least δ time, it deduces that the dwell time has expired and updates its state estimate accordingly.*

The final observer state is $z = (\{x_1'\}, y_b)$, which yields $C(\omega) = g(\{x_1'\}) = \{x_1\}$. Thus, the system is estimated to be in state x_1.

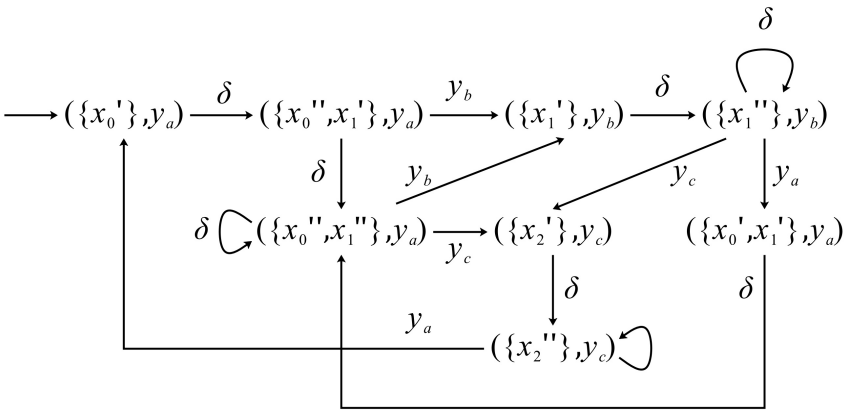


Figure 3.4: Observer for the power monitoring SOA from Example 3.1.

State Estimation Using the Observer. Once the observer has been constructed, it can be used to compute the set of states consistent with any observation sequence $\omega \in (Y, \mathbb{R}_{\geq\delta})^+$ by simply following the corresponding path in the observer.

Given an observer state $z = (X_z, y) \in Z$, we define $g(X_z) = \{x \in X \mid \{x', x''\} \cap X_z \neq \emptyset\}$ as the set of discrete states of the SOA that correspond to the extended discrete states in X_z. In other words, $g(X_z)$ extracts the base discrete states from the extended discrete states, ignoring the waiting/ready distinction.

When an observation sequence ω is observed, we follow the corresponding path in the observer, taking into account that events δ correspond to the elapsing of a time interval equal to δ while the current output does not change. Let $z = (X_z, y)$ be the observer state reached after following the observation ω. The set of states consistent with ω is then:

$$C(\omega) = g(X_z)$$

This provides an efficient online state estimation procedure: given an observation ω, we simply traverse the observer following the corresponding transitions, and the final observer state directly yields the set of consistent states $C(\omega)$.

Complexity Analysis. The number of states in the observer is bounded by the number of distinct simplified observer states. Since each observer state is of the form (X_z, y) where $X_z \subseteq X_e = X' \cup X''$ and $y \in Y$, and since the simplification procedure ensures that for each discrete state $x \in X$ at most one of x' or x'' is present in X_z, we have:

$$|Z| \leq 2^{|X|} \times |Y|$$

Therefore, the space complexity of the observer construction is $O(2^{|X|} \cdot |Y|)$. While this is exponential in the number of discrete states, it is significantly better than the naive bound of $O(2^{|Q_e|}) = O(2^{2|X||Y|})$ that would result without the state simplification procedure.

The time complexity of Algorithm 3 is $O(|Z| \cdot |\Sigma_e| \cdot |Q_e|)$, as for each of the $|Z|$ observer states, we consider each of the $|\Sigma_e|$ events, and for each event we perform operations whose cost is proportional to $|Q_e|$ (computing α and β functions).

3.4 Chapter Summary

This chapter presented the Switching Output Automaton framework for analyzing cyber-physical systems where internal states cannot be directly observed, but produce measurable output trajectories. The SOA model is a continuous-time discrete-event system featuring three distinguishing characteristics: set-valued output mappings that associate each state with multiple possible output values, output switching capabilities that allow

outputs to change independently of state transitions, and a minimal dwell time constraint requiring any system change (state transition or output switch) to be separated from the previous change by at least δ time units. This temporal constraint guarantees that only finitely many discrete events occur within any bounded time window.

When multiple states can generate identical outputs, external observers face fundamental uncertainty in determining the current system configuration. To address this challenge under incomplete observability, we developed two complementary finite-state constructions. Section 3.2 introduced the evolution automaton, which discretizes timing behavior by partitioning each global state into waiting and ready phases according to whether the dwell-time threshold has been satisfied. Section 3.3 constructed the observer through powerset-based determinization combined with a simplification procedure that exploits the waiting/ready structure, yielding an automaton that efficiently computes the set of states compatible with any given observation sequence.

These finite-state representations enable verification algorithms for security and reliability properties that depend on both discrete state information and temporal behavior. The observer provides the algorithmic mechanism for tracking system configurations under limited observation, supporting the opacity analysis in Chapter 4 and the fault diagnosis methods in Chapter 5.

Chapter 4

Opacity Verification in Switching Output Automata

4.1 Introduction

This chapter addresses the problem of opacity verification in Switching Output Automata, examining both state-based opacity (where secrets are discrete states) and timed opacity (where secrets depend on dwelling in global states for specific durations). We first establish efficient verification of current-state opacity using the observer construction from Chapter 3, then extend this framework to handle time-sensitive security properties that cannot be expressed using state-based opacity alone.

Section 4.2 addresses current-state opacity verification in the SOA framework. While classical opacity for NFAs was introduced in Chapter 2, the SOA model presents unique characteristics that necessitate adaptation: observations are continuous-time piecewise-constant signals rather than event sequences, and the dwell-time constraint introduces the notion of *stable observations*—those where the consistent state set no longer evolves with time elapse. We establish an efficient verification procedure by constructing an observer following the determinization approach from Chapter 3, then checking whether any stable observer state has a consistent state set contained entirely within the secret set. This procedure demonstrates how the observer construction enables algorithmic verification of security properties under partial observability.

While untimed opacity provides a foundation for state-based security analysis, it cannot capture information leakage that depends on temporal behavior. In many cyber-physical systems, the sensitivity of a state depends not only on which state is occupied, but on how long the system dwells there. For instance, in patient monitoring systems, a

brief yellow alert (5 minutes) may represent routine fluctuations that should remain private, whereas a sustained yellow alert (30 minutes) indicates deteriorating health that is operationally significant yet must be protected from unauthorized observers. The untimed framework cannot distinguish these scenarios: the secret set is either $X_s = \{\text{yellow\_alert}\}$ (protecting both cases) or $X_s = \emptyset$ (protecting neither). This fundamental limitation motivates the timed opacity framework developed in Section 4.3.

As discussed in Section 1.2, existing work on timed opacity focuses primarily on timed automata, where general verification is undecidable under dense-time semantics [14]. Decidability has been achieved only through severe restrictions such as limiting to real-time automata (RTA) with 2-EXPTIME complexity [15], [46], adopting discrete-time semantics at the model level [49], [50], or constraining the intruder's observational capabilities [53].

In contrast, the SOA framework—with its minimal dwell-time constraint—provides a natural setting where timed opacity becomes *decidable without the severe restrictions required for timed automata*. The minimal dwell time $\delta > 0$ bounds the rate of discrete transitions, enabling a finite discretization for verification purposes without restricting the expressiveness of the underlying continuous-time model. Moreover, the set-valued nature of the output mapping creates observational ambiguity independent of timing, allowing multiple states to share output symbols and thus enabling opacity.

Although SOA can be encoded in a clock-based formalism (e.g., by introducing a single clock c measuring time since the last change, with guard $c \geq \delta$ for any transition and reset $c := 0$ after the transition), our goal is to achieve decidable and implementable verification under partial observability. The minimal dwell-time constraint bounds the rate of changes and enables finite discretization (via evolution automata and observers), thereby avoiding the undecidability and complex symbolic machinery typically associated with dense-time opacity verification in general timed automata.

Our notion of timed opacity differs fundamentally from traditional opacity definitions in the timed automata literature that focus on whether certain states can be inferred from observations. We focus on *vulnerable states* paired with *secret dwell intervals*—configurations where the system remains in a state for a duration within a sensitive time range. For example, in hospital patient monitoring, sustained yellow alerts over clinically significant intervals (as defined by local protocols) may reveal treatment-response trajectories that should be protected under data protection regulations [82], [83]. In cyber-physical healthcare systems [84], such timing-based observation channels can leak information even when the underlying state and event sequences remain ambiguous.

This chapter establishes decidable verification of timed opacity for SOAs through a secret-dependent time discretization approach. We construct a finite evolution automaton

whose states represent configurations annotated with elapsed time, determinize it via observer construction, and reduce opacity verification to a finite-state reachability problem. The verification algorithm runs in time polynomial in the number of reachable observer states $|Z|$ and the size of the discretized state space $|\bar{Q}|$. In the worst case, $|Z|$ can be exponential in $|\bar{Q}|$ due to determinization. Formal notation, exact bounds, and proofs are provided in Sections 4.3.2 and 4.3.3, and Theorem 4.2. This discretization is made precise under the dwell-time quantization assumption introduced later.

The remainder of this chapter is organized as follows. Section 4.2 addresses untimed opacity verification in SOA. Section 4.3 formalizes timed secrets and opacity. Section 4.3.2 develops the secret-dependent evolution automaton with secret-dependent time discretization. Section 4.3.3 presents the observer-based verification algorithm and establishes the main decidability result. Case studies demonstrating the approach are presented in Chapter 6. Section 4.4 summarizes contributions and complexity analysis.

4.2 Untimed Opacity Verification

Recall from Definition 2.5 in Chapter 2 that a system is current-state opaque with respect to a secret set if an external observer cannot deduce with certainty that the system is currently in a secret state. We adapt this notion to the SOA framework.

Let $G = (X, Y, B, h, x_0, y_0)$ be an SOA as defined in Definition 3.1. Consider a subset $X_s \subseteq X$ of discrete states designated as *secret states*. An external intruder observes the system's piecewise-constant output signal $y(t)$ but cannot directly measure the discrete state $x(t)$. Given an observation sequence $\omega \in (Y \times \mathbb{R}_{\geq\delta})^+$, the observer constructed in Section 3.3 computes the set $C(\omega)$ of discrete states consistent with ω (as defined in Section 3.3).

In the SOA setting, where observations are continuous-time signals constrained by the minimal dwell time δ, the set of consistent states $C(\omega)$ may evolve as time elapses without output changes. Specifically, for observations of the form $\omega = \omega' \cdot (y, \tau)$ where the output has been y for duration τ, as τ increases the observer may transition through multiple states corresponding to different multiples of δ (since the evolution automaton uses δ as its fundamental time discretization unit and each δ-transition represents the passage of one minimal dwell time period). However, after a finite time, the observer reaches a stable state where further time elapse (without output change) does not modify $C(\omega)$.

Definition 4.1 (Stable Observations)**.** *An observation sequence* $\omega \in (Y \times \mathbb{R}_{\geq \delta})^+$ *is* stable *if the observer state* z *reached after processing* ω *does not enable the* δ *event, i.e.,* $\delta_o(z, \delta)$ *is not defined. The set of all stable observations is denoted* Ω_s*.*

Definition 4.2 (Current-State Opacity for SOA)**.** *An SOA* G *is* current-state opaque *with respect to a secret set* $X_s \subseteq X$ *if for all stable observations* $\omega \in \Omega_s$*, the set of consistent states satisfies* $C(\omega) \not\subseteq X_s$*.*

This definition ensures that an intruder, after observing a stable output sequence, cannot conclusively deduce that the system is currently in a secret state. The restriction to stable observations is motivated by the fact that non-stable observations represent transient estimations that become outdated within at most δ time units as time progresses.

To verify current-state opacity efficiently, we characterize stable observations in terms of the observer structure. Consider the observer $G_{\text{obs}} = (Z, \Sigma_e, \delta_o, z_0)$ constructed in Section 3.3, where $Z \subseteq 2^{X_e} \times Y$ and $\Sigma_e = Y \cup \{\delta\}$. Each observer state $z = (X_z, y) \in Z$ corresponds to a set $X_z \subseteq X_e$ of extended discrete states (waiting and ready states) and a current output $y \in Y$.

An observation ω is stable if and only if the corresponding observer state z reached after processing ω does not enable the δ event. This is because once no further δ-transitions are possible, the set of consistent states $C(\omega)$ cannot change unless a new output is observed. We formalize this as follows.

Definition 4.3 (Stable Observer States)**.** *A state* $z = (X_z, y) \in Z$ *of the observer is* stable *if* $\delta_o(z, \delta)$ *is not defined. The set of stable observer states is denoted* $Z_s = \{z \in Z \mid \delta_o(z, \delta)$ *is not defined*$\}$*.*

Stable observer states have the property that $X_z \subseteq X''$, i.e., they contain only ready extended states. This is because waiting states always enable a δ-transition to their corresponding ready states.

Recall from Section 3.3 that for an observer state $z = (X_z, y)$, the function $g(X_z) = \{x \in X \mid \{x', x''\} \cap X_z \neq \emptyset\}$ extracts the set of discrete states corresponding to the extended states in X_z. Using this notation, we can provide a necessary and sufficient condition for verifying current-state opacity.

Proposition 4.1 (Verification of Current-State Opacity)**.** *An SOA* G *is current-state opaque with respect to a secret set* $X_s \subseteq X$ *if and only if for all stable observer states* $z = (X_z, y) \in Z_s$*, it holds that* $g(X_z) \not\subseteq X_s$*.*

Proof. ($\Rightarrow$) Assume G is current-state opaque with respect to X_s. By Definition 4.2, for all stable observations $\omega \in \Omega_s$, we have $C(\omega) \not\subseteq X_s$. Each stable observation ω leads to

a stable observer state $z = (X_z, y) \in Z_s$ where $C(\omega) = g(X_z)$. Therefore, for all $z \in Z_s$, it holds that $g(X_z) \not\subseteq X_s$.

($\Leftarrow$) Assume there exists a stable observation $\omega \in \Omega_s$ such that $C(\omega) \subseteq X_s$. This observation leads to a stable observer state $z = (X_z, y) \in Z_s$ where $g(X_z) = C(\omega) \subseteq X_s$, contradicting the assumption. Therefore, if for all $z \in Z_s$ we have $g(X_z) \not\subseteq X_s$, then G is current-state opaque with respect to X_s. □

Proposition 4.1 provides an algorithmic verification procedure: construct the observer G_{obs} using Algorithm 3 from Chapter 3, identify the set Z_s of stable states (Definition 4.3), and check whether any stable state has $g(X_z) \subseteq X_s$. If such a state exists, the system is not opaque; otherwise, it is opaque.

Example 4.1 (Opacity Verification for SOA)**.** *Consider an SOA* $G = (X, Y, B, h, x_0, y_0)$ *with the following components:*

- *States:* $X = \{x_0, x_1, x_2\}$
- *Output alphabet:* $Y = \{y_a, y_b, y_c\}$
- *Arcs:* $B = \{(x_0, x_1), (x_1, x_2), (x_2, x_1)\}$
- *Output function:* $h(x_0) = \{y_a, y_b\}$, $h(x_1) = \{y_a, y_c\}$, $h(x_2) = \{y_b, y_c\}$
- *Initial state:* x_0*; Initial output:* $y_0 = y_a$

Let the secret set be $X_s = \{x_2\}$.

After constructing the observer following the procedure in Chapter 3, we identify the stable states in Z_s*. Among these stable states, suppose one state is* $z = (\{x_2''\}, y_b)$*, for which* $g(\{x_2''\}) = \{x_2\} = X_s$*. Since there exists a stable state whose consistent discrete states form a subset of the secret set, the system is not current-state opaque with respect to* $X_s = \{x_2\}$*. This is because an intruder observing the output sequence leading to state* z *can conclusively deduce that the system is currently in the secret state* x_2*.*

4.3 Timed Opacity Verification

This section develops the timed opacity framework where secrets depend on dwell durations in global states. Section 4.3.1 formalizes vulnerable states and secret dwell intervals, Section 4.3.2 constructs the secret-dependent evolution automaton with time discretization, and Section 4.3.3 presents the observer-based verification algorithm.

4.3.1 Problem Formulation

4.3.1.1 Vulnerable States and Secret Dwell Intervals

In many cyber-physical systems, certain operational configurations should remain hidden from external observers for security or privacy reasons. Unlike traditional opacity notions that focus on whether certain states can be inferred from observations, timed opacity is concerned with *how long* the system remains in specific configurations. For instance, in a smart grid system, a substation entering maintenance mode for a few minutes is routine testing, but remaining in that mode for 30–60 minutes indicates a potential equipment failure or security patch installation – information that adversaries could exploit to plan attacks during the vulnerable window.

We formalize this notion through vulnerable states and their associated secret dwell intervals.

Definition 4.4 (Vulnerable Global States)**.** *A set of* vulnerable global states *is a subset $Q_v \subseteq Q$ of the global state space. For each state $q \in Q_v$, remaining in q for specific time durations may reveal sensitive information about the system's behavior.*

Definition 4.5 (Secret Dwell Interval Mapping)**.** *Let $Q_v = \{q_1, q_2, \ldots, q_s\}$ be the set of vulnerable global states. A* secret dwell interval mapping *is a function*

$$\mathscr{I}_v : Q_v \to \{[a,b) : a \in \mathbb{R}_{\geq 0}, b \in (\mathbb{R}_{>a} \cup \{+\infty\})\}$$

that associates with each vulnerable state $q_i \in Q_v$ a left-closed, right-open interval $\mathscr{I}_v(q_i) = [\delta_i', \delta_i'')$, where:

- *$\delta_i' \in \mathbb{R}_{\geq 0}$ is the lower bound (minimal secret dwell time);*
- *$\delta_i'' \in \mathbb{R}_{>\delta_i'} \cup \{+\infty\}$ is the upper bound (maximal secret dwell time).*

To ensure compatibility with the discretization-based verification approach developed in subsequent sections, we impose a structural constraint on secret dwell intervals:

Assumption 4.1 (Interval Quantization)**.** *The lower and upper bounds of each secret dwell interval are integer multiples of the minimal dwell time δ (as defined in Definition 3.2). Specifically, for each $q_i \in Q_v$:*

$$\delta_i' = k_i' \delta, \quad \delta_i'' = k_i'' \delta, \quad k_i' \in \mathbb{N}_+, k_i'' \in \mathbb{N}_+ \cup \{+\infty\}, k_i' < k_i'',$$

where $\mathbb{N}_+ = \{1, 2, 3, \ldots\}$ denotes the set of positive integers.

This assumption is natural in practice, as the minimal dwell time δ defines the temporal resolution of the system, and secret intervals are typically specified as multiples of this fundamental time unit. Moreover, requiring $k_i' \geq 1$ ensures that secret intervals start at or after δ, meaning the system must remain in a vulnerable state for at least one minimal dwell time period before sensitive information is revealed.

Example 4.2 (Four-State SOA)**.** *Consider an SOA* $G = (X, Y, B, h, x_0, y_0)$ *with state set* $X = \{x_0, x_1, x_2, x_3\}$ *and output alphabet* $Y = \{1, 2, 3\}$*. The arc set is*

$$B = \{(x_0, x_1), (x_0, x_2), (x_1, x_3), (x_2, x_3), (x_3, x_2)\},$$

and the output function is defined by

$$\begin{aligned} h(x_0) &= \{1, 2\}, \quad h(x_1) = \{2\}, \\ h(x_2) &= \{1, 2\}, \quad h(x_3) = \{1, 3\}. \end{aligned}$$

The initial state is x_0 *and the initial output symbol is* $y_0 = 1$*. The set of direct successors are* $\sigma(x_0) = \{x_1, x_2\}$*,* $\sigma(x_1) = \{x_3\}$*,* $\sigma(x_2) = \{x_3\}$*, and* $\sigma(x_3) = \{x_2\}$*.*

The global state space is

$$Q = \{(x_0, 1), (x_0, 2), (x_1, 2), (x_2, 1), (x_2, 2), (x_3, 1), (x_3, 3)\}$$

with cardinality $|Q| = 7$*. Figure 4.1 shows the structure of this SOA.*

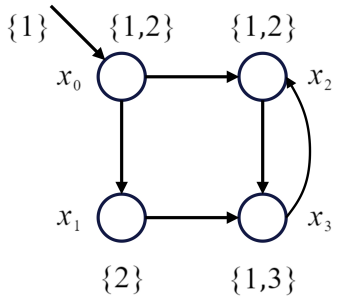


Figure 4.1: A simple four-state switching output automaton.

4.3.1.2 Timed Secret

With vulnerable states and secret dwell intervals defined, we now formally introduce the notion of a *timed secret*, which combines these two components to specify the configurations that must be protected from external observation.

Definition 4.6 (Timed Secret)**.** *Given a set of vulnerable global states Q_v and a secret dwell interval mapping $\mathscr{I}_v$, the* timed secret *is defined as*

$$S = S(Q_v, \mathscr{I}_v) = \{(q,t) \in Q \times \mathbb{R}_{\geq 0} \mid q \in Q_v, t \in \mathscr{I}_v(q)\}.$$

A pair $(q,t) \in S$ represents a secret configuration where the system has been in vulnerable global state q for duration t within the secret interval $\mathscr{I}_v(q)$.

The timed secret S thus specifies the set of *configurations to be protected* from external observation. The goal is to ensure that an observer monitoring only the system outputs cannot determine whether the current configuration (q,t) belongs to S.

Example 4.3 (Secret Definition for Four-State SOA)**.** *Consider the SOA shown in Figure 4.1. We define the set of vulnerable global states as $Q_v = \{(x_2,2)\}$, with secret dwell interval $\mathscr{I}_v((x_2,2)) = [2\delta,4\delta)$, giving the secret $S = \{((x_2,2),t) \mid t \in [2\delta,4\delta)\}$.*

4.3.1.3 Timed Opacity

The timed secret specifies which configurations must be protected. We now formally define what it means for an SOA to be *opaque* with respect to such a secret: an external observer, monitoring only the timed output sequence, cannot determine whether the system is currently in a secret configuration.

Definition 4.7 (Timed Opacity)**.** *An SOA $G = (X,Y,B,h,x_0,y_0)$ is* opaque *with respect to a timed secret S if an external observer is never able to determine with certainty whether the current configuration (q,t) – where $q \in Q$ is the current global state and t is the dwell time since the most recent entry into q – belongs to S.*

More precisely, timed opacity holds if and only if: for every observation sequence $\omega \in L_\delta(G)$, if there exists a run ρ generating ω such that the configuration reached belongs to S, then there must exist another run ρ' also generating ω such that the configuration reached does not belong to S.

With timed opacity formally defined, we turn to the verification problem: given an SOA G and a timed secret S, determine whether G is opaque with respect to S. The following sections develop an automata-theoretic approach that renders this verification problem *decidable* through secret-dependent time discretization and observer construction.

4.3.2 The Secret-Dependent Evolution Automaton

To determine whether an SOA can protect a timed secret from observation, we construct a finite-state abstraction that tracks the dwell time progression in each global state through discrete logical states. This abstraction, called the *secret-dependent evolution automaton*, yields a finite representation suitable for automated verification.

The fundamental challenge is that the timed secret S is defined over continuous time: a configuration (q,t) belongs to S if the system has remained in q for a duration t within a continuous interval. Because standard automata-theoretic verification techniques operate on finite-state structures, we discretize time while preserving exactly the distinctions needed to determine whether secret configurations can be observationally separated from non-secret ones.

4.3.2.1 Motivation for Time Discretization

Consider the vulnerable global state $q = (x_2, 2)$ with secret dwell interval $\mathscr{I}_v(q) = [2\delta, 4\delta)$ from Example 4.3. To verify opacity, we need to track:

- Whether the system is currently in state q;
- How long the system has been in state q;
- Whether this duration falls within the secret interval.

The dwell time is a continuous variable, yet we need a *finite* representation. The key observation is that we do not need to know the exact dwell time—only which interval it belongs to. Specifically, for opacity verification, we need to distinguish:

1. **Before minimal dwell time**: $t \in [0, \delta)$ — no transitions can occur yet;
2. **Before secret interval**: $t \in [\delta, 2\delta)$ — system is in q but not in secret;
3. **Within secret interval**: $t \in [2\delta, 4\delta)$ — system configuration belongs to S;
4. **Beyond secret interval**: $t \in [4\delta, +\infty)$ — system is in q but no longer in secret.

This motivates a discretization where each global state is split into a sequence of *logical states*, each representing a specific time interval.

4.3.2.2 Formal Definition of Logical States

Although an observer can measure exact dwell times through output observations (as timed output symbols (y, τ) defined in Section 3.1.3), for the purpose of opacity verification, we do not need to track all possible real-valued durations in our automaton model. Instead, we only need to distinguish which critical time interval a dwell time belongs to—specifically, whether the system is before, within, or beyond the secret dwell interval. This insight allows us to partition the continuous time dimension into a finite sequence of intervals for each global state. The granularity of this discretization depends on whether the global state is vulnerable:

- For *non-vulnerable states*, we only need to distinguish whether the minimal dwell time δ has elapsed (this is necessary to track when transitions become enabled, even though no secret is involved);
- For *vulnerable states*, we must track the progression through the secret interval with finer granularity, creating distinct intervals that capture entry into, duration within, and exit from the secret dwell interval.

A *logical state* $(x,y)_j$ represents configurations in which the global state is (x,y) and the dwell time belongs to a specific interval I_j. We introduce a function $\mathscr{R} : Q \to \mathbb{N}_+$ that maps each global state $q \in Q$ to the maximum interval index (and hence the maximum logical state index) for that state. The definition of $\mathscr{R}(q)$ depends on whether q is vulnerable:

Non-vulnerable global states. If $q = (x,y) \notin Q_v$, we set $\mathscr{R}(q) = 1$. The intervals of interest are:

$$I_0 = [0, \delta) \quad \text{and} \quad I_1 = [\delta, +\infty)$$

corresponding to the logical state sequence

$$(x,y)_0 \xrightarrow{\delta} (x,y)_1 \circlearrowleft_{\delta}$$

This creates two logical states: $(x,y)_0$ represents the initial interval before the minimal dwell time has elapsed, and $(x,y)_1$ represents all subsequent time with a self-loop for indefinite dwelling.

Vulnerable global states. If $q = (x,y) \in Q_v$ is vulnerable with secret dwell interval $\mathscr{I}_v(q) = [k'\delta, k''\delta)$, we define:

$$\mathscr{R}(q) = \begin{cases} k'' & \text{if } k'' \in \mathbb{N} \\ \max\{1,k'\} & \text{if } k'' = +\infty \end{cases}$$

The second case ensures finiteness: when the secret interval is unbounded ($k'' = +\infty$), we discretize time only up to the start of the secret interval, as distinguishing between different durations beyond $k'\delta$ is unnecessary for opacity verification.

For vulnerable states with finite upper bound ($k'' \in \mathbb{N}$), let $k = \mathscr{R}(q) = k''$. The intervals of interest are:

$$I_0 = [0,\delta), I_1 = [\delta,2\delta), \ldots, I_{k-1} = [(k-1)\delta, k\delta), I_k = [k\delta, +\infty)$$

corresponding to the logical state sequence

$$(x,y)_0 \xrightarrow{\delta} (x,y)_1 \xrightarrow{\delta} \ldots \xrightarrow{\delta} (x,y)_{k-1} \xrightarrow{\delta} (x,y)_k \circlearrowleft_\delta .$$

This creates $k+1$ logical states, with δ-transitions between consecutive states and a self-loop on the final state $(x,y)_k$.

For vulnerable states with unbounded upper bound ($k'' = +\infty$), let $k = \mathscr{R}(q) = \max\{1,k'\}$. The intervals of interest are:

$$I_0 = [0,\delta), I_1 = [\delta,2\delta), \ldots, I_{k-1} = [(k-1)\delta, k\delta), I_k = [k\delta, +\infty)$$

corresponding to the logical state sequence

$$(x,y)_0 \xrightarrow{\delta} (x,y)_1 \xrightarrow{\delta} \ldots \xrightarrow{\delta} (x,y)_k \circlearrowleft_\delta .$$

This creates $k+1$ logical states. The unique secret logical state is $(x,y)_k$; its δ-self-loop represents all dwell times $t \geq k\delta$.

Characterization of logical states. The following properties hold for all logical states:

- **Initial state restriction:** From logical state $(x,y)_0$ (i.e., before the minimal dwell time δ elapses), no discrete state transitions or output changes may occur (this applies to every newly entered global state). This enforces the minimal dwell time constraint of the SOA.

- **Transition availability:** From all logical states $(x,y)_j$ where $j > 0$, transitions to different discrete states or output changes are possible (subject to the behavior relation B and output function h).

- **Secret logical states:** For a vulnerable global state $q = (x,y) \in Q_v$ with secret interval $\mathscr{I}_v(q) = [k'\delta, k''\delta)$, the logical states $(x,y)_j$ with indices $j \in \{k', k'+1, \ldots, k''-1\}$ are called *secret logical states*. These correspond precisely to configurations where the SOA is in global state q with dwell time $t \in [k'\delta, k''\delta)$, meaning $(q,t) \in S$.

Table 4.1 summarizes the discretization scheme for different types of global states.

Global State Type	$\mathscr{R}(q)$	**# Logical States**	**Secret State Indices**
Non-vulnerable ($q \notin Q_v$)	1	2	None
Vulnerable with $[k'\delta, k''\delta)$, $k'' \in \mathbb{N}$	k''	$k''+1$	$\{k', k'+1, \ldots, k''-1\}$
Vulnerable with $[k'\delta, +\infty)$	$\max\{1, k'\}$	$\max\{1, k'\}+1$	$\{k'\}$ (if $k' > 0$)

Table 4.1: Summary of logical state discretization scheme for different global state types.

Example 4.4 (Logical States for Four-State SOA)**.** *Consider the four-state SOA from Example 4.2 with the secret defined in Example 4.3. Recall that $Q_v = \{(x_2,2)\}$ with secret interval $\mathscr{I}_v((x_2,2)) = [2\delta, 4\delta)$. We compute the logical states for each global state:*

Non-vulnerable global states: *For all global states $q \notin Q_v$, we have $\mathscr{R}(q) = 1$, yielding two logical states each:*

- $(x_0,1)$*: logical states* $(x_0,1)_0$ *and* $(x_0,1)_1$
- $(x_0,2)$*: logical states* $(x_0,2)_0$ *and* $(x_0,2)_1$
- $(x_1,2)$*: logical states* $(x_1,2)_0$ *and* $(x_1,2)_1$
- $(x_2,1)$*: logical states* $(x_2,1)_0$ *and* $(x_2,1)_1$
- $(x_3,1)$*: logical states* $(x_3,1)_0$ *and* $(x_3,1)_1$
- $(x_3,3)$*: logical states* $(x_3,3)_0$ *and* $(x_3,3)_1$

Vulnerable global state: *For the vulnerable state $q = (x_2,2) \in Q_v$ with secret interval $\mathscr{I}_v(q) = [2\delta, 4\delta)$, we have $k' = 2$ and $k'' = 4$. Since $k'' \in \mathbb{N}$, we get $\mathscr{R}(q) = k'' = 4$. This yields 5 logical states:*

$$(x_2,2)_0, (x_2,2)_1, (x_2,2)_2, (x_2,2)_3, (x_2,2)_4$$

The time evolution for this vulnerable state is:

$$(x_2,2)_0 \xrightarrow{\delta} (x_2,2)_1 \xrightarrow{\delta} \underbrace{(x_2,2)_2 \xrightarrow{\delta} (x_2,2)_3}_{\textit{secret logical states}} \xrightarrow{\delta} (x_2,2)_4 \circlearrowleft_{\delta}$$

The secret logical states *are those with indices* $j \in [k',k'') = [2,4)$*:*

$$\{(x_2,2)_2,(x_2,2)_3\}$$

4.3.2.3 Structure of the Evolution Automaton

Building on the logical state discretization, we now formally define the secret-dependent evolution automaton. This automaton provides a finite-state representation that captures all possible behaviors of the SOA at the level of abstraction required for opacity verification. The key idea is to replace each global state (x,y) of the original SOA with a sequence of logical states $(x,y)_0,(x,y)_1,\ldots,(x,y)_{\mathscr{R}(q)}$ that explicitly track the progression of dwell time through discrete intervals.

Definition 4.8 (Secret-Dependent Evolution Automaton)**.** *Given an SOA* $G = (X,Y,B,h,x_0,y_0)$ *and a timed secret* $S = \{(q_i,t) \in Q \times \mathbb{R}_{\geq 0} \mid q_i \in Q_v, t \in \mathscr{I}_v(q_i)\}$*, the* secret-dependent evolution automaton *is a nondeterministic finite automaton* $G_e(S) = (\bar{Q},Y_e,\Delta,\bar{q}_0)$ *where*

- $\bar{Q} = \{(x,y)_j \mid q = (x,y) \in Q,\ j \in \{0,\ldots,\mathscr{R}(q)\}\}$ *is a finite set of logical states;*
- $Y_e = Y \cup \{\delta\}$ *is the alphabet;*
- $\Delta \subseteq \bar{Q} \times (Y_e \cup \{\varepsilon\}) \times \bar{Q}$ *is the transition relation;*
- $\bar{q}_0 = (x_0,y_0)_0$ *is the initial state.*

Logical state set $\bar{Q}$. The state space is constructed by expanding each global state $q = (x,y) \in Q$ of the original SOA into $\mathscr{R}(q)+1$ logical states. By Proposition 4.2, $\bar{Q}$ is finite since Q is finite and each $\mathscr{R}(q)$ is bounded. The cardinality of $\bar{Q}$ depends on the secret specification: more fine-grained secret intervals lead to larger state spaces.

Extended alphabet Y_e. The automaton evolves over the extended observable alphabet $Y_e = Y \cup \{\delta\}$, where the δ-symbol denotes one minimal dwell-time period. This allows the automaton to explicitly model time progression as discrete events. Transitions labeled with $y \in Y$ represent observable output changes (output switching events), while δ-labeled transitions represent time elapsing without output changes.

Transition relation Δ. The transition relation encodes the possible state changes in the evolution automaton, including time-elapsing events, unobservable state changes, and observable output changes. The detailed classification of transition types is presented in Section 4.3.2.4.

Initial state $\bar{q}_0$. The system starts in logical state $(x_0, y_0)_0$, with index $j = 0$ indicating that no time has elapsed yet. This reflects the fact that at time $t = 0$, the system has just entered its initial global state and the dwell time counter starts from zero.

Definition 4.9 (Secret Interval Bounds)**.** *For any vulnerable global state $q = (x,y) \in Q_v$ with $\mathscr{I}_v(q) = [k'\delta, k''\delta)$, set*

$$\mathscr{R}'(q) = k', \qquad \mathscr{R}''(q) = \begin{cases} k'' - 1, & k'' \in \mathbb{N}, \\ k', & k'' = +\infty. \end{cases}$$

The corresponding secret logical states are

$$\mathscr{L}_{secret}(q) = \{(x,y)_j \mid j \in [\mathscr{R}'(q), \mathscr{R}''(q)] \cap \mathbb{N}\},$$

and the global secret set is $S_v = \bigcup_{q \in Q_v} \mathscr{L}_{secret}(q)$.

Notation. For brevity we occasionally write $\mathscr{R}(x,y)$ instead of $\mathscr{R}(q)$, where $q = (x,y)$.

4.3.2.4 Transition Types

The evolution automaton employs four types of transitions to model the timed behavior of the SOA: time-elapsing transitions (δ-labeled) that track duration within logical states, and three types of event-based transitions (Type 1, 2, 3) that model state and output changes. Together, these transitions capture all possible behaviors of the original SOA at the discretized abstraction level. We now formally characterize each type.

Time-elapsing transitions (δ-labeled). A δ-labeled transition represents time elapsing by one δ-time unit. Such a transition from a logical state $(x,y)_j$ yields the logical state $(x,y)_{j+1}$ if $j < \mathscr{R}(x,y)$, representing progression to the next time interval. When $j = \mathscr{R}(x,y)$, the transition becomes a self-loop $(x,y)_j \xrightarrow{\delta} (x,y)_j$, indicating that the system remains in the final time interval indefinitely.

Event-based transitions. In addition to time elapsing, from a logical state $(x,y)_j$ when $j > 0$ (i.e., after the minimal dwell time has elapsed), three types of event-based transitions may occur that model state and output changes:

- **Type 1 (State change with no output change):** When the SOA transitions from discrete state x to $\bar{x} \neq x$ while the output y remains unchanged (according to the behavior relation B), an external observer cannot detect its occurrence. This means that the system transitions from the global state (x,y) to the global state $(\bar{x},y)$, and the dwell time of $(\bar{x},y)$ starts from zero. In the evolution automaton we denote this by a transition labeled with the empty string ε from a logical state $(x,y)_j$ to another logical state $(\bar{x},y)_0$.

- **Type 2 (Simultaneous state and output change):** When the SOA transitions from global state (x,y) to $(\bar{x},\bar{y})$ where $x \neq \bar{x}$ and $y \neq \bar{y}$, both the discrete state and output change simultaneously. The dwell time of $(\bar{x},\bar{y})$ starts from zero. We denote this by a transition labeled $\bar{y}$ from $(x,y)_j$ to $(\bar{x},\bar{y})_0$.

- **Type 3 (Output change with no state change):** When the SOA transitions from global state (x,y) to $(x,\bar{y})$ where $y \neq \bar{y}$, only the output changes while the discrete state remains the same. The dwell time of $(x,\bar{y})$ starts from zero. We denote this with a transition labeled $\bar{y}$ from $(x,y)_j$ to $(x,\bar{y})_0$.

When a transition of type 2 or 3 occurs, the output changes from y to $\bar{y}$ and thus an external observer detects that a transition has occurred. However, due to the possibility that different discrete states may produce the same output (i.e., their output sets have nonempty intersection), the external observer may not be able to determine whether a change in the discrete state has also occurred.

Table 4.2 summarizes the characteristics of each transition type.

Transition Type	**Label**	**State Changes?**	**Output Changes?**	**Observable?**
Time elapsing	δ	No	No	Yes (duration)
Type 1	ε	Yes ($x \to \bar{x}$)	No	No
Type 2	$\bar{y} \in Y$	Yes ($x \to \bar{x}$)	Yes ($y \to \bar{y}$)	Yes
Type 3	$\bar{y} \in Y$	No	Yes ($y \to \bar{y}$)	Yes

Table 4.2: Summary of transition types in the evolution automaton.

4.3.2.5 Construction Algorithm

We now present a systematic procedure for constructing $G_e(S)$ from a given SOA and timed secret. The construction follows a reachability-based approach: starting from the initial logical state, the algorithm explores all reachable logical states by iteratively generating transitions according to the four types defined in Section 4.3.2.4. This exploration ensures that the resulting automaton contains precisely the logical states and transitions necessary to represent all possible timed behaviors of the SOA.

Algorithm 4 explores all reachable logical states starting from the initial logical state $\bar{q}_0 = (x_0, y_0)_0$. For each logical state $\bar{q} = (x,y)_j$, it generates time-elapsing δ-transitions (lines 6–13) and, when $j > 0$ (enforcing minimal dwell time), the three types of event-based transitions: Type 1 (unobservable state changes, lines 15–22), Type 2 (state and output changes, lines 23–29), and Type 3 (output-only changes, lines 30–36).

Example 4.5 (Evolution Automaton for Four-State SOA)**.** *Consider the four-state SOA from Example 4.2 with secret S from Example 4.3. Applying Algorithm 4, we construct the secret-dependent evolution automaton* $G_e(S) = (\bar{Q}, Y_e, \Delta, \bar{q}_0)$.

The initial logical state is $\bar{q}_0 = (x_0,1)_0$*, and the extended alphabet is* $Y_e = \{1,2,3,\delta\}$. *Following Example 4.4, the logical state set* $\bar{Q}$ *contains:*

- *For each non-vulnerable global state: 2 logical states (e.g.,* $(x_0,1)_0, (x_0,1)_1$*)*
- *For the vulnerable state* $(x_2,2)$*: 5 logical states* $(x_2,2)_0, \ldots, (x_2,2)_4$

The transition relation Δ *includes:*

- ***δ-transitions****: For example,* $(x_0,1)_0 \xrightarrow{\delta} (x_0,1)_1$ *(line 7 of the algorithm) and* $(x_0,1)_1 \xrightarrow{\delta} (x_0,1)_1$ *(self-loop, line 12)*
- ***Type 1 (ε-transitions)****: For instance, from* $(x_0,1)_1$*, since* $\sigma(x_0) = \{x_1,x_2\}$ *and* $1 \in h(x_2) = \{1,2\}$*, we have* $(x_0,1)_1 \xrightarrow{\varepsilon} (x_2,1)_0$ *(lines 15–18)*
- ***Type 2 (state and output change)****: From* $(x_0,1)_1$*, since* $2 \in h(x_0) = \{1,2\}$ *and* $\sigma(x_0) = \{x_1,x_2\}$ *with* $2 \in h(x_1)$ *and* $2 \in h(x_2)$*, we have transitions like* $(x_0,1)_1 \xrightarrow{2} (x_1,2)_0$ *and* $(x_0,1)_1 \xrightarrow{2} (x_2,2)_0$ *(lines 23–26)*
- ***Type 3 (output-only change)****: From* $(x_0,1)_1$*, since* $2 \in h(x_0) \setminus \{1\}$*, we have* $(x_0,1)_1 \xrightarrow{2} (x_0,2)_0$ *(lines 30–33)*

The resulting evolution automaton $G_e(S)$ *is nondeterministic. For instance, logical state* $(x_0,1)_1$ *can activate the transition labeled* 2*, leading to three different logical states:* $(x_0,2)_0$*,* $(x_1,2)_0$*, and* $(x_2,2)_0$*. Additionally, it can activate* ε*-transition to reach* $(x_2,1)_0$.

Algorithm 4 Constructing the secret-dependent evolution automaton

Input: $G = (X, Y, B, h, x_0, y_0)$, a switching output automaton and $S = \{(q_i, t) \in Q \times \mathbb{R}_{\geq 0} \mid q_i \in Q_v, t \in \mathscr{I}_v(q_i)\}$, the timed secret
Output: $G_e(S) = (\bar{Q}, Y_e, \Delta, \bar{q}_0)$, the secret-dependent evolution automaton

Set $\bar{q}_0 = (x_0, y_0)_0$, $\bar{Q}_{\text{new}} = \{\bar{q}_0\}$, $\bar{Q} = \emptyset$
Set $Y_e = Y \cup \{\delta\}$
Set $\Delta = \emptyset$
while $\bar{Q}_{\text{new}} \neq \emptyset$ **do**
    select a state $\bar{q} = (x, y)_j \in \bar{Q}_{\text{new}}$
    Let $q = (x, y)$ be the corresponding global state
    if $j \in \{0, 1, \ldots, \mathscr{R}(q) - 1\}$ **then**
        $\bar{q}' = (x, y)_{j+1}$, $\Delta = \Delta \cup \{(\bar{q}, \delta, \bar{q}')\}$
        if $\bar{q}' \notin \bar{Q}_{\text{new}} \cup \bar{Q}$ **then**
            $\bar{Q}_{\text{new}} = \bar{Q}_{\text{new}} \cup \{\bar{q}'\}$
        end if
    end if
    if $j = \mathscr{R}(q)$ **then**
        $\Delta = \Delta \cup \{(\bar{q}, \delta, \bar{q})\}$
    end if
    if $j > 0$ **then** ▷ Type 1/2/3 transitions are enabled only for $j > 0$ to enforce the minimal dwell time in any state
        for all $\bar{x} \in \sigma(x)$ **do**
            if $y \in h(\bar{x})$ **then**
                $\bar{q}' = (\bar{x}, y)_0$, $\Delta = \Delta \cup \{(\bar{q}, \varepsilon, \bar{q}')\}$
                if $\bar{q}' \notin \bar{Q}_{\text{new}} \cup \bar{Q}$ **then**
                    $\bar{Q}_{\text{new}} = \bar{Q}_{\text{new}} \cup \{\bar{q}'\}$
                end if
            end if
            for all $\bar{y} \in h(\bar{x}) \setminus \{y\}$ **do**
                $\bar{q}' = (\bar{x}, \bar{y})_0$, $\Delta = \Delta \cup \{(\bar{q}, \bar{y}, \bar{q}')\}$
                if $\bar{q}' \notin \bar{Q}_{\text{new}} \cup \bar{Q}$ **then**
                    $\bar{Q}_{\text{new}} = \bar{Q}_{\text{new}} \cup \{\bar{q}'\}$
                end if
            end for
        end for
        for all $\bar{y} \in h(x) \setminus \{y\}$ **do**
            $\bar{q}' = (x, \bar{y})_0$, $\Delta = \Delta \cup \{(\bar{q}, \bar{y}, \bar{q}')\}$
            if $\bar{q}' \notin \bar{Q}_{\text{new}} \cup \bar{Q}$ **then**
                $\bar{Q}_{\text{new}} = \bar{Q}_{\text{new}} \cup \{\bar{q}'\}$
            end if
        end for
    end if
    $\bar{Q}_{\text{new}} = \bar{Q}_{\text{new}} \setminus \{\bar{q}\}$, $\bar{Q} = \bar{Q} \cup \{\bar{q}\}$
end while

Figure 4.2 shows the complete structure of this evolution automaton, illustrating all logical states and transitions including δ-transitions (time elapse), ε-transitions (unobservable state changes), and observable transitions labeled with output symbols.

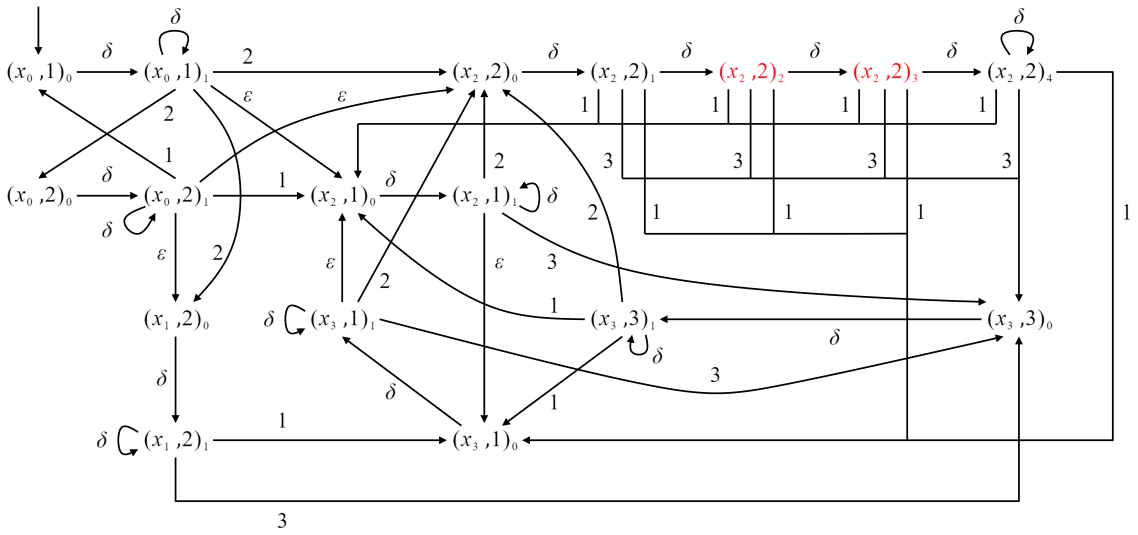


Figure 4.2: The secret-dependent evolution automaton $G_e(S)$ for the four-state switching output automaton.

4.3.2.6 Properties of the Evolution Automaton

We now establish key properties of the evolution automaton that justify its use for timed opacity verification. These properties ensure that the discretization preserves the essential behavioral characteristics of the original SOA while providing a finite-state representation suitable for automated analysis.

Proposition 4.2 (Finiteness)**.** *For any SOA $G = (X, Y, B, h, x_0, y_0)$ with finite X and Y, and any timed secret S (with finite Q_v and intervals quantized by δ), the evolution automaton $G_e(S)$ has a finite state set $\bar{Q}$.*

Proof. For each global state $q = (x, y) \in Q$, the function $\mathscr{R}(q)$ yields a finite maximum index:

- If $q \notin Q_v$, then $\mathscr{R}(q) = 1$, yielding logical states $(x, y)_0$ and $(x, y)_1$.
- If $q \in Q_v$ and $\mathscr{I}_v(q) = [k'\delta, k''\delta)$ with $k'' \in \mathbb{N}$, then $\mathscr{R}(q) = k''$; if $\mathscr{I}_v(q) = [k'\delta, +\infty)$, then $\mathscr{R}(q) = \max\{k', 1\}$.

Since both X and Y are finite, the global state space $Q = \{(x,y) \mid x \in X, y \in h(x)\}$ is finite with $|Q| \leq |X| \cdot |Y|$. Each $q \in Q$ contributes at most $\mathscr{R}(q)+1$ logical states, so the total number of logical states is:

$$|\bar{Q}| = \sum_{q \in Q} (\mathscr{R}(q)+1) < \infty$$

Thus $\bar{Q}$ is finite, and since $\Delta \subseteq \bar{Q} \times (Y_e \cup \{\varepsilon\}) \times \bar{Q}$, the transition relation Δ is also finite. □

Proposition 4.3 (Correspondence with SOA Runs)**.** *Let $G = (X, Y, B, h, x_0, y_0)$ be an SOA and $G_e(S)$ its evolution automaton. For any run ρ of G, there exists a corresponding path π in $G_e(S)$ such that the observable output sequence of ρ matches the output-labeled transitions of π.*

Proof sketch. The correspondence is established by construction. Every state-output run (x_i, y_i, t_i) of the SOA corresponds to a sequence of logical states in $G_e(S)$:

1. While in global state (x_i, y_i) for time t_i, the evolution automaton traverses logical states $(x_i, y_i)_0 \xrightarrow{\delta} (x_i, y_i)_1 \xrightarrow{\delta} \cdots$ until the appropriate time interval is reached. Each δ-transition corresponds to one minimal dwell time period elapsed, which is observable to an external observer monitoring the system output duration.

2. When the SOA transitions from (x_i, y_i) to (x_j, y_j) (either by discrete state change or output switch), the evolution automaton performs a corresponding Type 1, 2, or 3 transition from some $(x_i, y_i)_k$ to $(x_j, y_j)_0$.

The observable behavior is preserved as follows:

- **Observable symbols:** Both δ-transitions (representing time elapsing) and output-labeled transitions (representing output changes) are observable. An external observer can detect when the output remains constant for duration t (abstracted as $\lfloor t/\delta \rfloor$ occurrences of δ) and when the output changes from y to y'.

- **Unobservable transitions:** Only ε-transitions (Type 1 transitions representing internal state changes without output changes) remain unobservable, as they do not produce any observable event.

□

Proposition 4.4 (Complexity Bounds)**.** *Let $n_Q = |Q|$, $n_Y = |Y|$, $k_{\max} = \max_{q \in Q_v} \mathscr{R}(q)$, and $\sigma_{\max} = \max_{x \in X} |\sigma(x)|$. Then*

1. ***State space:*** $|\bar{Q}| \leq n_Q\,(k_{\max}+1)$.

2. ***Transition relation:*** *Every* $(x,y)_j \in \bar{Q}$ *satisfies*

$$\deg^+\big((x,y)_j\big) \;\leq\; 1 \;+\; |\sigma(x)| \;+\; (|\sigma(x)|+1)\,n_Y,$$

hence

$$|\Delta| \;\leq\; |\bar{Q}| \cdot \big[\,1+\sigma_{\max}+(\sigma_{\max}+1)\,n_Y\,\big].$$

Proof. (1) As before, each global state contributes at most $\mathscr{R}(q)+1 \leq k_{\max}+1$ logical states.

(2) Any logical state has at most one δ-transition. Type 1 (ε) transitions contribute at most $|\sigma(x)|$. Type 2 transitions emanate to states $(\bar{x},\bar{y})_0$ where $\bar{x} \in \sigma(x)$ and $\bar{y} \in h(\bar{x})$, contributing at most $|\sigma(x)| \cdot n_Y$ transitions. Type 3 transitions emanate to states $(x,\bar{y})_0$ where $\bar{y} \in h(x)$, contributing at most n_Y transitions. Summing yields $1+|\sigma(x)|+|\sigma(x)| \cdot n_Y + n_Y = 1+|\sigma(x)|+(|\sigma(x)|+1)n_Y$. Replacing $|\sigma(x)|$ with $\sigma_{\max}$ and summing over all logical states completes the claim. □

Remark 4.1. *The degree bound is conservative: it treats each source state as emitting up to n_Y symbols. Replacing n_Y by $|h(\bar{x})|$ for Type 2 transitions and $|h(x) \setminus \{y\}|$ for Type 3 transitions yields instance-specific tighter bounds.*

4.3.3 Observer-Based Verification

The secret-dependent evolution automaton $G_e(S)$ provides a finite-state abstraction of the SOA's timed behavior. We now develop a verification methodology to determine whether the system satisfies timed opacity. The verification approach is based on constructing an observer automaton that tracks the possible states the system could be in based on observations, and then checking whether any observation sequence uniquely reveals that the system is in a secret configuration.

To connect the continuous-time observations from the SOA with the discrete symbolic sequences processed by the evolution automaton, we formalize the observation abstraction function.

Definition 4.10 (Observation Abstraction Function)**.** *Given an SOA $G=(X,Y,B,h,x_0,y_0)$, we define the function $\psi : L_\delta(G) \to Y_e^*$ as follows:*

$$\psi((y,t)) = \delta^k \quad \textit{where } k = \lfloor t/\delta \rfloor$$
$$\psi(\omega(y,t)) = \psi(\omega)\,y\,\delta^k \quad \textit{where } k = \lfloor t/\delta \rfloor$$

where $\lfloor\cdot\rfloor$ denotes the floor function, $\omega \in L_\delta(G)$, and $Y_e^* = (Y \cup \{\delta\})^*$ denotes the set of all finite sequences over $Y_e = Y \cup \{\delta\}$.

Remark 4.2 (Granularity safety of ψ)**.** *Under Assumption 4.1, membership in any secret dwell interval $[k'\delta, k''\delta)$ depends only on the number of completed δ-quanta. Therefore replacing a duration t by $\lfloor t/\delta \rfloor$ δ-symbols is sound for opacity verification: sub-δ remainders do not change whether a dwell time has crossed a secret boundary.*

Note. Observation strings have the shape $\delta^{k_0} y_1 \delta^{k_1} y_2 \delta^{k_2} \cdots$; the initial output y_0 is not emitted explicitly, and a new output symbol appears only when the output value changes. The initial output value is carried by the initial logical state $(x_0, y_0)_0$, hence no explicit emission of y_0 is needed; a new output symbol is emitted only upon changes.

Example 4.6 (Output Behavior Abstraction)**.** *Consider the SOA in Figure 4.1. We set the minimal dwell time $\delta = 1$ and suppose the output behavior is $\omega = (1, 2.5)(2, 3)(3, 1.6)$. We compute the corresponding logical observation sequence:*

$$s = \psi(\omega) = \delta^2\, 2\, \delta^3\, 3\, \delta.$$

4.3.3.1 Observer Construction

Definition 4.11 (Observer)**.** *Given an evolution automaton $G_e(S) = (\bar{Q}, Y_e, \Delta, \bar{q}_0)$, the* observer *of $G_e(S)$ is a deterministic automaton $G_{obs} = (Z, Y_e, \Delta_o, z_0)$ where:*

- *$Z \subseteq 2^{\bar{Q}}$ is a finite set of observer states, where each state $z \in Z$ is a subset of logical states from $\bar{Q}$;*
- *$Y_e = Y \cup \{\delta\}$ is the alphabet (same as the evolution automaton);*
- *$\Delta_o : Z \times Y_e \to Z$ is a partial deterministic transition function (undefined for unreachable state-symbol pairs);*
- *$z_0 = \{(x_0, y_0)_0\}$ is the initial observer state, containing only the initial logical state.*

The transition function Δ_o extends to sequences $s \in Y_e^*$ in the standard recursive manner: $\Delta_o(z, \varepsilon) = z$ and $\Delta_o(z, y_e s) = \Delta_o(\Delta_o(z, y_e), s)$ where $y_e \in Y_e$ and $s \in Y_e^*$.

Reachability functions. To construct the observer, we define auxiliary functions that compute reachable states in the evolution automaton:

- **ε-closure:** For any logical state $\bar{q} \in \bar{Q}$, we define

$$D_\varepsilon(\bar{q}) = \{\bar{q}' \in \bar{Q} \mid \exists m \geq 0,\ \bar{q} = \bar{q}_0 \xrightarrow{\varepsilon} \bar{q}_1 \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} \bar{q}_m = \bar{q}'\},$$

  i.e., the set of states reachable from $\bar{q}$ by a (possibly empty) path consisting *only* of ε-transitions. By definition, $\bar{q} \in D_\varepsilon(\bar{q})$.

- **Observable transition:** For any logical state $\bar{q} \in \bar{Q}$ and symbol $y \in Y_e$, we define

$$D_y(\bar{q}) = \{\bar{q}' \in \bar{Q} \mid (\bar{q}, y, \bar{q}') \in \Delta\}$$

  as the set of all logical states reachable from $\bar{q}$ by executing exactly one observable y-transition.

- **Set-based reachability:** For any set of logical states $\bar{Q}_{\text{sub}} \subseteq \bar{Q}$ and symbol $y \in Y_e$, we define:

$$\alpha(\bar{Q}_{\text{sub}}, y) = \bigcup_{\bar{q} \in \bar{Q}_{\text{sub}}} D_y(\bar{q})$$
$$\beta(\bar{Q}_{\text{sub}}, y) = \bigcup_{\bar{q} \in \alpha(\bar{Q}_{\text{sub}}, y)} D_\varepsilon(\bar{q})$$

  where $\alpha(\bar{Q}_{\text{sub}}, y)$ computes states reachable by one y-transition from any state in $\bar{Q}_{\text{sub}}$, and $\beta(\bar{Q}_{\text{sub}}, y)$ further includes all states reachable via subsequent ε-transitions.

Observer construction algorithm. Algorithm 5 systematically constructs the observer by exploring all reachable observer states.

Example 4.7 (Observer Construction for Four-State SOA)**.** *Applying Algorithm 5 to the evolution automaton $G_e(S)$ from Example 4.5 produces the observer $G_{obs} = (Z, Y_e, \Delta_o, z_0)$ where:*

- *The initial observer state is $z_0 = D_\varepsilon(\{(x_0, 1)_0\}) = \{(x_0, 1)_0\}$*
- *The extended alphabet is $Y_e = \{1, 2, 3, \delta\}$*
- *Observer states are subsets of logical states from $\bar{Q}$*

For example, processing the observation δ from z_0 yields:

$$\Delta_o(z_0, \delta) = \beta(\{(x_0, 1)_0\}, \delta) = \{(x_0, 1)_1\}$$

Algorithm 5 Constructing the observer automaton

Input: $G_e(S) = (\bar{Q}, Y_e, \Delta, \bar{q}_0)$
Output: $G_{\text{obs}} = (Z, Y_e, \Delta_o, z_0)$
$z_0 = D_\varepsilon(\bar{q}_0)$, $Z = \{z_0\}$, mark z_0 as unprocessed
while there exists an unprocessed $z \in Z$ **do**
    pick an unprocessed z
    for each $y \in Y_e$ **do**
        $\alpha(z,y) = \bigcup_{\bar{q}\in z} D_y(\bar{q})$
        $\beta(z,y) = \bigcup_{\bar{q}\in\alpha(z,y)} D_\varepsilon(\bar{q})$
        $\bar{z} = \beta(z,y)$
        if $\bar{z} \neq \emptyset$ **then**
            if $\bar{z} \notin Z$ **then**
                $Z \leftarrow Z \cup \{\bar{z}\}$, mark $\bar{z}$ as unprocessed
            end if
            $\Delta_o(z,y) = \bar{z}$ ▷ Partial function: only defined for nonempty successors
        end if
    end for
    mark z as processed
end while

From this state, observing output 2 *leads to:*

$$\Delta_o(\{(x_0,1)_1\},2) = \beta(\{(x_0,1)_1\},2) = \{(x_0,2)_0,(x_1,2)_0,(x_2,2)_0\}$$

This observer state contains three logical states because the nondeterministic evolution automaton has three possible 2*-labeled transitions from* $(x_0,1)_1$*. Figure 4.3 shows the complete observer structure.*

Properties of the observer. The constructed observer satisfies several important correctness properties that ensure its suitability for opacity verification.

Proposition 4.5 (State Estimate Correctness)**.** *Let* $G_e(S) = (\bar{Q}, Y_e, \Delta, \bar{q}_0)$ *be an evolution automaton and* $G_{obs} = (Z, Y_e, \Delta_o, z_0)$ *its observer. For any sequence* $s \in Y_e^*$*, the observer state* $z = \Delta_o(z_0, s)$ *correctly estimates the set of reachable logical states:* z *contains all and only those logical states in* $\bar{Q}$ *that are reachable in* $G_e(S)$ *from* $\bar{q}_0$ *by a path whose observable projection is* s*.*

Proof sketch. The correctness follows by induction on the length of s, using the construction of α and β functions. The base case ($s = \varepsilon$) holds by initialization: $z_0 = D_\varepsilon(\bar{q}_0)$ contains all and only states reachable via ε-transitions. The inductive step follows from the correctness of the β computation, which accounts for both observable transitions and subsequent ε-closure. □

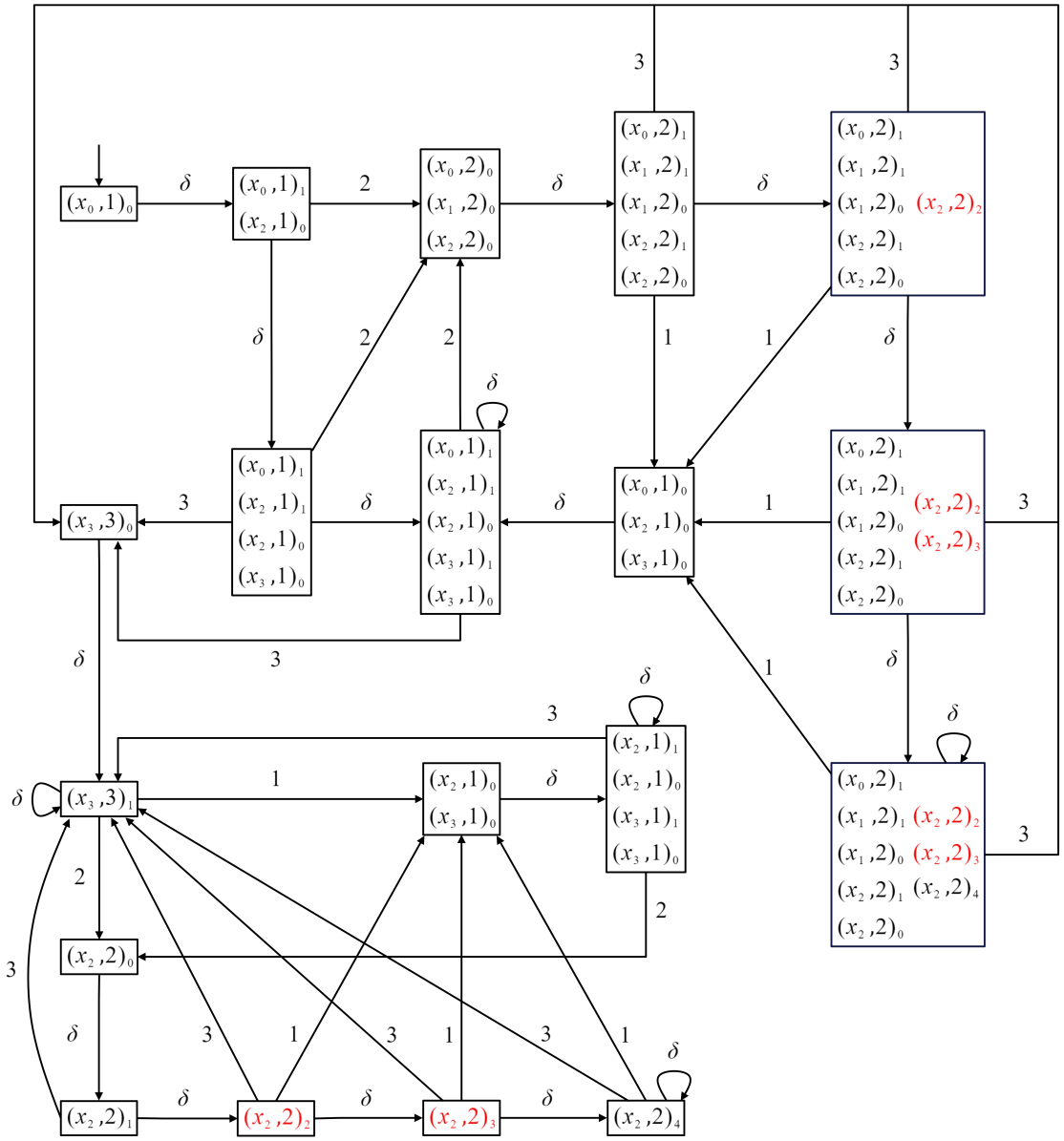


Figure 4.3: The observer automaton G_{obs} for the four-state switching output automaton.

Proposition 4.6 (Observer Determinism)**.** *The observer automaton G_{obs} is deterministic: for any state $z \in Z$ and symbol $y \in Y_e$, there exists at most one successor state $\Delta_o(z,y)$.*

Proof. By construction, Algorithm 5 computes a unique successor $\bar{z} = \beta(z,y)$ for each pair (z,y) and assigns $\Delta_o(z,y) = \bar{z}$. The determinism follows directly from the functional nature of β. □

Complexity analysis. The observer construction has the following complexity characteristics:

- **State space:** In the worst case, $|Z| \leq 2^{|\bar{Q}|}$, as each observer state is a subset of $\bar{Q}$. However, in practice, only a subset of the powerset is reachable, often much smaller than the theoretical maximum.
- **Time complexity:** Algorithm 5 explores at most $|Z|$ states. For each state z and each symbol $y \in Y_e$, computing $\beta(z,y)$ requires:
  - Computing $\alpha(z,y)$: $O(|z| \cdot |\Delta|)$ to find all y-transitions from states in z.
  - Computing $\beta(z,y)$: $O(|\alpha(z,y)| \cdot |\bar{Q}|^2)$ for ε-closure computation (using transitive closure or BFS).

  Overall time complexity: $O(|Z| \cdot |Y_e| \cdot |\bar{Q}|^3)$ in the worst case.
- **Space complexity:** $O(|Z| \cdot |\bar{Q}|)$ to store observer states and $O(|Z| \cdot |Y_e|)$ for transitions.

For the four-state SOA with $|\bar{Q}| = 17$ logical states, the theoretical maximum observer size is $2^{17} \approx 131$ thousand states, but the actual reachable observer typically has only dozens of states due to the structure of the evolution automaton.

4.3.3.2 Opacity Verification Condition

With both the secret-dependent evolution automaton $G_e(S)$ and its deterministic observer G_{obs} constructed, we now establish the formal condition for verifying timed opacity. We first formalize the notion of secret logical states introduced in Section 4.3.2.

Definition 4.12 (Secret Logical States)**.** *For a vulnerable global state $q = (x,y) \in Q_v$ with secret interval $\mathscr{I}_v(q) = [k'\delta, k''\delta)$, the set of* secret logical states *is:*

$$\mathscr{L}_{secret}(q) = \{(x,y)_j \mid j \in [\mathscr{R}'(q), \mathscr{R}''(q)] \cap \mathbb{N}\}$$

where $\mathscr{R}'(q) = k'$ *and* $\mathscr{R}''(q) = k'' - 1$ *for finite intervals, or* $\mathscr{R}''(q) = k'$ *for unbounded intervals* $[k'\delta, +\infty)$.

Definition 4.13 (Secret Logical State Set)**.** *The* secret logical state set *is:*

$$S_v = \bigcup_{q \in Q_v} \mathscr{L}_{secret}(q)$$

where $q = (x, y)$.

Theorem 4.1 (Timed Opacity Verification Condition)**.** *Let* $G = (X, Y, B, h, x_0, y_0)$ *be an SOA,* S *a timed secret,* $G_e(S) = (\bar{Q}, Y_e, \Delta, \bar{q}_0)$ *its secret-dependent evolution automaton, and* $G_{obs} = (Z, Y_e, \Delta_o, z_0)$ *the observer of* $G_e(S)$*. The system* G *is timed opaque with respect to* S *if and only if:*

$$\forall z \in Z : z \not\subseteq S_v$$

Proof. We prove both directions of the equivalence.

(⇒) Necessity: Assume G is timed opaque with respect to S. We prove by contradiction that $\forall z \in Z : z \not\subseteq S_v$.

Suppose, for the sake of contradiction, that there exists an observer state $z^* \in Z$ such that $z^* \subseteq S_v$. Since $z^* \in Z$ is reachable in the observer automaton, there exists a sequence $s \in Y_e^*$ such that:

$$z^* = \Delta_o(z_0, s)$$

By Proposition 4.5 (State Estimate Correctness), z^* contains all and only those logical states in $\bar{Q}$ that are reachable in $G_e(S)$ from $\bar{q}_0$ by paths whose observable projection is s. In particular, since $z^* \neq \emptyset$ (observer states are always nonempty by construction), there exists at least one logical state $\bar{q} \in z^*$ reachable via observable sequence s.

By the correspondence between evolution automaton paths and SOA runs (Proposition 4.3), there exists a run ρ of the SOA G and an observation sequence $\omega \in L_\delta(G)$ such that:

1. The abstraction of ω equals s, i.e., $\psi(\omega) = s$;
2. The run ρ reaches a configuration (q, t) corresponding to logical state $\bar{q}$.

Now, since $z^* \subseteq S_v$ and $\bar{q} \in z^*$, we have $\bar{q} \in S_v$. By Definition 4.13, this means $(q, t) \in S$ is a secret configuration.

Moreover, since $z^* \subseteq S_v$, *every* logical state in z^* is a secret logical state. This means that *all* configurations of G consistent with observation ω belong to the secret set S. Formally:

$$\{(q', t') \mid (q', t') \text{ is consistent with observation } \omega\} \subseteq S$$

By Definition 4.7, this violates timed opacity: the observer can deduce with certainty that the current configuration belongs to S after observing ω. This contradicts our assumption that G is timed opaque.

Therefore, no such observer state z^* can exist, and we conclude $\forall z \in Z : z \not\subseteq S_v$.

($\Leftarrow$) Sufficiency: Assume $\forall z \in Z : z \not\subseteq S_v$. We prove that G is timed opaque with respect to S.

Let ω be any observation sequence generated by the SOA G, and let $s = \psi(\omega) \in Y_e^*$ be its abstraction. The observer processes s and reaches state:

$$z = \Delta_o(z_0, s)$$

By Proposition 4.5, z contains exactly those logical states reachable in $G_e(S)$ via paths with observable projection s. By our assumption, $z \not\subseteq S_v$, which means there exists at least one logical state $\bar{q} \in z$ such that $\bar{q} \notin S_v$.

By Definition 4.13, $\bar{q} \notin S_v$ means that $\bar{q}$ corresponds to a non-secret configuration $(q,t) \notin S$ of the SOA. By Proposition 4.3, there exists a run of G producing observation ω that passes through configuration (q,t).

Since $(q,t) \notin S$ is consistent with observation ω, the observer cannot determine with certainty that all configurations consistent with ω belong to S. Therefore, for every observation ω, there exists at least one consistent configuration that is not secret, satisfying the timed opacity condition (Definition 4.7).

Thus, G is timed opaque with respect to S. □

Remark 4.3. *The verification condition can be efficiently checked by integrating the subset test $z \subseteq S_v$ into Algorithm 5. During observer construction, check each newly discovered observer state z; if $z \subseteq S_v$, the system violates opacity. If all reachable observer states satisfy $z \not\subseteq S_v$, the system is timed opaque.*

Example 4.8 (Timed Opacity Verification for Four-State SOA)**.** *Consider the system in Figure 4.1 with its observer constructed in Example 4.7. Only $(x_2,2)$ is a vulnerable global state, hence from Example 4.3 we have $\mathscr{R}'((x_2,2)) = 2$ and $\mathscr{R}''((x_2,2)) = 3$. The set of secret logical states is:*

$$S_v = \{(x_2,2)_2, (x_2,2)_3\}.$$

Applying Theorem 4.1, we examine whether any observer state $z \in Z$ satisfies $z \subseteq S_v$. Among the observer states, we find that $z_1 = \{(x_2,2)_2\}$ and $z_2 = \{(x_2,2)_3\}$ both satisfy

$z \subseteq S_v$:

$$z_1 = \{(x_2, 2)_2\} \subseteq S_v, \quad z_2 = \{(x_2, 2)_3\} \subseteq S_v.$$

Therefore, by Theorem 4.1, the system is not *timed opaque with respect to the secret S. This means there exist observation sequences that allow an external observer to deduce with certainty that the system has been in state* $(x_2, 2)$ *for a duration within the secret interval* $[2\delta, 4\delta)$.

Proposition 4.7 (Verification Complexity)**.** *The verification procedure has time complexity* $O(|Z| \cdot |Y_e| \cdot |\bar{Q}|^3)$*, where the additional subset checks* $z \subseteq S_v$ *require* $O(|z|) = O(|\bar{Q}|)$ *time per observer state and do not increase the asymptotic complexity.*

Proof. Observer construction explores at most $|Z|$ states, processing $|Y_e|$ symbols per state. Computing $\beta(z, y)$ requires $O(|\bar{Q}|^3)$ time (for α computation and ε-closure). The subset check $z \subseteq S_v$ takes $O(|\bar{Q}|)$ time, dominated by β computation. Total: $O(|Z| \cdot |Y_e| \cdot |\bar{Q}|^3)$. □

Theorem 4.2 (Overall Complexity)**.** *Let* $G = (X, Y, B, h, x_0, y_0)$ *be an SOA and let S be a timed secret. Define* $k_{\max} := \max_{q \in Q_v} \mathscr{R}(q)$ *and* $T_{\max} := \delta\, k_{\max}$*. The end-to-end verification pipeline satisfies:*

1. *Evolution automaton construction (Algorithm 4) runs in time*

$$O\Big(|\bar{Q}| \cdot \big[\, 1 + \sigma_{\max} + (\sigma_{\max} + 1)\, n_Y \,\big]\Big),$$

   with $|\bar{Q}| \leq n_Q\,(k_{\max} + 1)$*. When* $|Y|$ *is bounded and* $\sigma_{\max} = O(|X|)$*, this simplifies to* $O\big(|X|^2 \cdot (T_{\max}/\delta)\big)$.

2. *Observer-based verification (Algorithm 5) runs in time* $O(|Z| \cdot |Y_e| \cdot |\bar{Q}|^3)$ *and space* $O(|Z| \cdot |\bar{Q}|)$ *(Proposition 4.7), where* $|\bar{Q}| \leq n_Q\,(k_{\max} + 1)$ *and* $T_{\max} = \delta\, k_{\max}$.

Worst-case note. *While the end-to-end runtime is polynomial in* $|\bar{Q}|$ *and the number of reachable observer states* $|Z|$*, the latter can be exponential in* $|\bar{Q}|$ *in the worst case due to determinization (powerset) construction.*

Proof sketch. Item (1) follows from the out-degree bound in Proposition 4.4, because Algorithm 4 visits each reachable logical state once and emits at most $1 + \sigma_{\max} + (\sigma_{\max} + 1)\, n_Y$ outgoing transitions per state. Item (2) follows from Proposition 4.7 combined with the cardinality bound $|\bar{Q}| \leq n_Q\,(k_{\max} + 1)$. The relationship $T_{\max} = \delta\, k_{\max}$ is by definition of the discretization depth. □

Practical scalability. We emphasize that the worst-case exponential blowup in opacity verification stems from observer-based determinization and is inherent to opacity verification under partial observability—even for untimed discrete-event systems, opacity verification is known to be PSPACE-complete in general. In the SOA setting, timed opacity introduces an additional abstraction layer by discretizing dwell time; however, the discretization is *secret-dependent* rather than uniform, so the abstract state size $|\bar{Q}|$ scales primarily with the vulnerable-state set and the time-window specification, not with the entire system and continuous time domain. Moreover, our constructions are on-the-fly and explore only the *reachable* portion of the observer state space, which is often far smaller than the theoretical powerset bound. Finally, SOA can reduce model-level inflation compared to DES encodings that emulate dwell-time behavior by unfolding time into additional states or injecting timer events—common workarounds that can severely harm scalability. The case studies in Chapter 6 report the resulting automata sizes (including thousand-state-scale determinized structures in the battery diagnosis example), and Chapter 7 outlines compositional and symbolic extensions to further scale to industrial systems.

4.4 Chapter Summary

This chapter established decidable verification procedures for opacity properties in Switching Output Automata, addressing both state-based secrets and time-dependent secrets where sensitivity depends on how long the system remains in vulnerable configurations.

Section 4.2 developed verification for current-state opacity by introducing the notion of stable observer states as the verification criterion. This approach avoids checking all possible observations by focusing only on those where state estimates have converged, reducing verification to checking whether any stable observer state contains exclusively secret states. Section 4.3 formalized timed opacity where secrets are global-state-duration triples $(q,t) = ((x,y),t)$ rather than states alone, capturing scenarios where brief and prolonged occupancy of the same global state have different security implications. Section 4.3.2 introduced secret-dependent time discretization that applies fine-grained quantization only to vulnerable global states while using coarse quantization elsewhere, transforming the infinite-state continuous-time problem into finite-state verification without losing precision for security-critical timing behavior. Section 4.3.3 established the main decidability result: timed opacity holds if and only if no reachable observer state consists entirely of secret logical states, providing a necessary and sufficient condition checkable through finite-state exploration.

The framework achieves decidability for timed opacity in SOAs without the restrictions required for timed automata, where verification is generally undecidable or requires limiting to syntactic subclasses with high complexity. The complexity analysis in Theorem 4.2 shows that while observer construction may incur exponential blowup in the worst case due to determinization, the actual reachable observer size in practical examples remains tractable. This work establishes the theoretical foundation for automated verification of time-sensitive security properties in cyber-physical systems.

Chapter 5

Timed Fault Diagnosis in Switching Output Automata

5.1 Introduction

Fault diagnosis in time-sensitive systems requires detecting anomalies whose occurrence depends on how long the system dwells in critical configurations. Existing timed diagnosis frameworks based on timed automata and time Petri nets [76], [77] rely on zone-based state exploration or symbolic reachability analysis, facing PSPACE-complete verification complexity and scalability challenges due to dense-time clock dynamics. Tick-event discretization approaches [79] achieve decidability by introducing explicit time-advance events, but this augmentation leads to state-space explosion when fine-grained temporal resolution is needed. These limitations motivate alternative modeling paradigms that capture time-dependent fault dynamics while maintaining computational tractability.

The Switching Output Automaton (SOA) framework from Chapter 4 enforces a uniform minimal dwell time $\delta > 0$ across all system evolutions, yielding a natural temporal granularity for verification. To model time-dependent failures within this framework, we introduce the *Switching Output Automaton with Faults* (SOAF)—an extension that partitions the transition relation into *nominal arcs* representing intended system behavior and *fault arcs* representing degradation or failure modes. Unlike nominal transitions that remain enabled indefinitely once δ has elapsed, fault arcs are governed by *time-gated enablement windows*: they activate only when the continuous dwell time in a fault-prone state falls within specified intervals. This architecture reflects physical reality where failures emerge from prolonged exposure to stress conditions rather than instantaneous events; moreover, the discrete δ-time granularity enables finite-state abstraction without

artificial discretization, yielding a finite reachability graph that makes verification computationally tractable.

Traditional fault diagnosis detects discrete fault events—sensor failures, actuator malfunctions, or component breakdowns that occur at specific instants. The SOAF framework targets a different failure class: *residence-time-triggered faults* that manifest only after sustained dwell in vulnerable configurations. Battery thermal runaway exemplifies this phenomenon: merely entering a high-temperature state poses no immediate danger, but remaining above 60°C for extended periods (e.g., beyond 15 minutes) initiates exothermic reactions leading to catastrophic failure. Similarly, power semiconductor junctions degrade through thermomechanical stress accumulation, with mean time to failure dropping exponentially as sustained junction temperature increases. Event-based fault models cannot capture these degradation processes; they require explicit representation of vulnerability windows tied to state residence durations. The SOAF framework provides this capability through its time-gated fault arc semantics.

Verification proceeds through a four-stage finite-state abstraction pipeline. First, we discretize continuous dwell time into integer multiples of δ. Second, we construct an *Evolution Automaton with Faults* (EAF) that tracks both discrete state and discretized dwell time. Third, a *fault recognizer* is built by marking which state-time configurations are consistent with observed output sequences and their durations, yielding a nondeterministic automaton whose states carry labels indicating whether the system trajectory is normal, faulty, or uncertain. Fourth, applying standard subset construction produces a *deterministic diagnoser* that processes observations online, outputting `NORMAL`, `FAULTY`, or `UNCERTAIN` verdicts with bounded delay after fault occurrence. Recognizer construction scales linearly with the size of the evolution automaton; determinization incurs worst-case exponential blowup characteristic of observer-based diagnosis [10]. This finite-state construction ensures decidability of fault diagnosis by reducing the problem to reachability analysis in the diagnoser automaton (Proposition 5.4). The battery management case study in Chapter 6 illustrates detection delay analysis for thermal runaway faults triggered by dwelling in high-temperature states.

The remainder of this chapter proceeds as follows. Section 5.2 formalizes the problem setting, introducing fault-prone states (Section 5.2.1), fault time mapping that encodes enablement windows (Section 5.2.2), and the complete SOAF model definition (Section 5.2.3). Section 5.3 presents the four-stage diagnosis pipeline architecture, followed by detailed construction procedures: logical state discretization (Section 5.3.1), Evolution Automaton with Faults construction (Section 5.3.2), fault recognizer synthesis (Section 5.3.3), and deterministic diagnoser construction (Section 5.3.4). Chapter 6 validates the framework on a battery thermal management case study where thermal runaway

faults are triggered by dwelling in high-temperature states beyond critical thresholds. Section 5.4 summarizes key results and discusses extensions.

5.2 Problem Formulation

We focus on dwell-time–triggered faults: a fault becomes enabled only when the current dwell time since the most recent entry into a global state $q = (x, y)$ lies within specified intervals. This captures degradation phenomena driven by sustained exposure to stress conditions (e.g., overheating in power electronics, electrolyte decomposition in batteries, or structural stress in mechanical systems). The dwell timer resets whenever the global state changes (either the discrete state x or the output y), reflecting that brief interruptions allow partial recovery (e.g., cooling dissipates accumulated heat).

Remark 5.1 (Scope and Extensions)**.** *This framework models within-episode faults governed by dwell time. Lifetime cumulative aging effects (e.g., capacity fade in batteries, creep in metals) can be incorporated via parameterized fault windows $\mathscr{I}_\theta$ where $\theta \in [0,1]$ represents a health parameter (e.g., State of Health): as θ decreases, the windows shrink to reflect accelerated degradation. In the core formalism (Sections 5.2.1–5.3.4), fault windows are fixed; the battery case study in Chapter 6 illustrates health-dependent parameterization.*

Given an SOA with time-gated fault arcs, the problem is to decide from timed output observations whether a fault has occurred, immediately or within a bounded delay. This entails: (1) extending SOA with fault arcs and enablement windows; (2) constructing a δ-quantized evolution automaton that preserves window boundaries; and (3) synthesizing a deterministic diagnoser for *normal/faulty/uncertain* classification.

The Switching Output Automaton with Faults (SOAF) extends the SOA with *fault-prone states* X_f (where faults may originate), *fault arcs* B_f (transitions representing fault occurrences), and *fault time mappings* $\mathscr{I}$ (specifying when faults are enabled during a state's dwell).

5.2.1 Fault-Prone States and Fault Arcs

5.2.1.1 Fault-Prone States

In the context of fault diagnosis, not all system states are equally susceptible to failures. Certain operational configurations expose the system to conditions that may trigger faults after dwelling for specific time durations. We formalize this notion through fault-prone states.

Definition 5.1 (Fault-Prone States)**.** *A* fault-prone state *is a discrete state $x \in X$ of an SOA from which the system may experience a fault after dwelling for specific time durations. The set of all fault-prone states is denoted:*

$$X_f \subseteq X$$

Remark 5.2. *In practical systems, X_f is typically a strict subset of X (i.e., $X_f \subsetneq X$), reflecting that only certain operational modes are vulnerable to time-dependent failures.*

5.2.1.2 Fault Arcs

Definition 5.2 (Fault Arcs)**.** *The set of* fault arcs *is denoted $B_f \subseteq X_f \times X$, representing transitions that model fault occurrences. Each fault arc $b_f = (x, x') \in B_f$ is an ordered pair from a fault-prone state $x \in X_f$ to a successor state $x' \in X$. Fault arcs are disjoint from nominal behavior ($B_f \cap B = \emptyset$) and can fire only inside specified time windows (defined later).*

Each $b_f = (x, x')$ induces $|h(x)| \cdot |h(x')|$ global transitions $(x, y) \to (x', y')$ with $y \in h(x)$ and $y' \in h(x')$.

Fault successors. Analogous to $\sigma(x)$ for nominal transitions, we define:

Definition 5.3 (Fault Successors)**.** *For a fault-prone state $x \in X_f$, the set of* fault successors *is:*

$$\sigma_f(x) = \{x' \in X \mid \exists b_f = (x, x') \in B_f\}$$

This represents all states reachable from x via fault transitions.

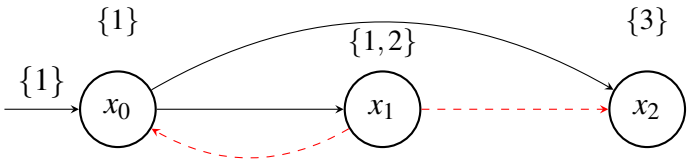


Figure 5.1: An SOA with fault arcs.

Example 5.1 (SOA with Fault Arcs)**.** *Consider the SOA $G = (X, Y, B, h, x_0, y_0)$ illustrated in Figure 5.1, with state set $X = \{x_0, x_1, x_2\}$, output alphabet $Y = \{1, 2, 3\}$, nominal arcs $B = \{(x_0, x_1), (x_0, x_2)\}$, and output function $h(x_0) = \{1\}$, $h(x_1) = \{1, 2\}$, $h(x_2) = \{3\}$.*

We extend this base SOA by defining fault arcs $B_f = \{(x_1, x_0), (x_1, x_2)\}$, which are marked in red in the figure. Note that state x_1 has no outgoing nominal transitions (i.e.,

there is no arc $(x_1, x') \in B$ for any $x' \in X$). The fault arcs represent potential unintended transitions that are disjoint from the nominal behavior ($B_f \cap B = \emptyset$). From these definitions, we can derive:

- *The set of fault-prone states is $X_f = \{x_1\}$ (only x_1 has outgoing fault arcs).*
- *The set of fault successors from x_1 is $\sigma_f(x_1) = \{x_0, x_2\}$.*

This structural extension specifies which *fault transitions are possible but does not yet constrain* when *they can occur during a dwell in state x_1. The temporal dimension will be addressed through the fault time mapping in the next section.*

5.2.2 Fault Time Mapping

Notation on δ. Following Chapter 4, δ denotes the minimal dwell time. When time is discretized later, δ also serves as the unit step for dwell counters; the same symbol is kept by convention.

The fault time mapping specifies, for each fault-prone global state (x, y) and each outgoing fault arc b_f, the precise time intervals during which the fault arc becomes active.

Definition 5.4 (Fault-Prone Global States)**.** *The set of* fault-prone global states *is:*

$$Q_f = \{(x, y) \mid x \in X_f, y \in h(x)\}$$

representing all state-output pairs from which fault transitions may originate.

The distinction between global states is essential because the same discrete state $x \in X_f$ may exhibit different failure dynamics under different observable conditions $y \in h(x)$. For instance, when a fault-prone state produces multiple outputs $h(x) = \{y_1, y_2, \ldots\}$ corresponding to different observable conditions (e.g., temperature ranges, voltage levels), each resulting global state (x, y_i) may have distinct temporal vulnerabilities. A motor operating at high temperature may fail faster than the same motor at normal temperature, even though both conditions correspond to the same discrete operational mode.

Definition 5.5 (Fault Time Mapping)**.** *For $x \in X_f$, let $B_f(x) := \{(x, x') \in B_f \mid x' \in X\}$ denote the set of all fault arcs originating from x. A* fault time mapping *is a partial function*

$$\mathscr{I} : D \to \mathrm{Fin}^+(\mathrm{Intv})$$

where $D \subseteq \{(q, b_f) \mid q = (x, y) \in Q_f,\ b_f \in B_f(x)\}$ is the effective domain of definition, $\mathrm{Intv} := \{[a, b) \mid a \in \mathbb{R}_{\geq 0},\ b \in \mathbb{R}_{>0} \cup \{+\infty\},\ a < b\}$*, and* $\mathrm{Fin}^+(\mathrm{Intv})$ *denotes all finite*

non-empty subsets of Intv. *For each* $(q, b_f) \in D$,

$$\mathscr{I}(q, b_f) = \{[\delta'_1, \delta''_1), [\delta'_2, \delta''_2), \ldots, [\delta'_n, \delta''_n)\}$$

with $n \in \mathbb{N}_+$. *The intervals are pairwise disjoint and temporally ordered:*

$$\delta'_1 < \delta''_1 < \delta'_2 < \delta''_2 < \cdots < \delta'_n < \delta''_n \leq +\infty,$$

and for all m we have $\delta'_m < \delta''_m$. *We require a minimal dwell-time constraint* $\delta'_1 \geq \delta$. *We adopt half-open intervals* $[a, b)$ *so that faults are disabled exactly at b and no boundary double-enabling occurs.*

Intuitively, $\mathscr{I}(q, b_f)$ returns a finite set of pairwise-disjoint windows during which b_f is enabled while the system dwells in q; if $(q, b_f) \notin D$, the fault never occurs from q. The dwell time t is measured since the most recent entry into the global state (x, y); the timer resets whenever the global state changes, including when the output changes within the same discrete state.

Multiple disjoint intervals. A fault arc may have multiple enabling windows modeling independent triggering mechanisms (e.g., thermal stress at $[10\delta, 15\delta)$ and bearing wear at $[45\delta, 60\delta)$) or nonlinear degradation (early and late vulnerability phases separated by a stable period). Adjacent windows with $\delta''_m = \delta'_{m+1}$ are merged canonically.

Minimal dwell time constraint. Faults require $\delta'_1 \geq \delta$ (fault arcs respect the same minimal dwell as nominal arcs). Unlike nominal transitions (enabled indefinitely after δ), fault arcs have bounded windows capturing time-dependent degradation.

Assumption 5.1 (Interval Quantization for Faults)**.** *The lower and upper bounds of each fault occurrence interval satisfy:*

- *For all* $1 \leq m \leq n$: $\delta'_m = k'_m \delta$ *where* $k'_m \in \mathbb{N}_+$ *(positive integers), ensuring* $\delta'_1 \geq \delta$ *(lower bounds are positive integer multiples of* δ*);*
- *For all* $1 \leq m < n$: $\delta''_m = k''_m \delta$ *where* $k''_m \in \mathbb{N}_+$ *(intermediate upper bounds are positive integer multiples of* δ*);*
- *For* $m = n$: *either* $\delta''_n = k''_n \delta$ *where* $k''_n \in \mathbb{N}_+$, *or* $\delta''_n = +\infty$ *(the last upper bound may be infinite, representing persistent vulnerability after a threshold).*

This quantization assumption serves two purposes:

1. **Practical realism:** In practice, fault timing specifications are typically given as multiples of a fundamental time unit (e.g., "failure occurs between 10 and 15 minutes of operation"), aligning with how engineers specify mean time to failure (MTTF) and related metrics.

2. **Discrete abstraction:** Using δ as the fundamental time unit enables the construction of finite-state abstractions (the Evolution Automaton with Faults, introduced in Section 5.3.2) that discretize continuous time while preserving the essential temporal structure of fault occurrences.

To maintain a canonical, non-redundant specification, adjacent windows (i.e., with $\delta''_m = \delta'_{m+1}$) should be merged into a single interval.

Example 5.2 (Comprehensive Fault Time Mapping)**.** *Consider the SOA with fault arcs from Example 5.1, where* $X_f = \{x_1\}$, $B_f = \{(x_1,x_0),(x_1,x_2)\}$, *and* $h(x_1) = \{1,2\}$. *By Definition 5.4, the fault-prone global states are* $Q_f = \{(x_1,1),(x_1,2)\}$. *We define the following fault occurrence intervals that satisfy Assumption 5.1 (illustrated in Figure 5.2):*

- $\mathscr{I}((x_1,1),(x_1,x_0)) = \{[\delta,2\delta),[4\delta,5\delta)\}$*: Two disjoint intervals for the fault from* x_1 *to* x_0 *when output is* 1.

- $\mathscr{I}((x_1,2),(x_1,x_0)) = \{[2\delta,3\delta),[6\delta,+\infty)\}$*: An initial bounded interval and an unbounded interval starting at* 6δ.

- $\mathscr{I}((x_1,2),(x_1,x_2)) = \{[3\delta,4\delta)\}$*: A single bounded interval for the fault from* x_1 *to* x_2 *when output is* 2.

Note that there is no mapping for $((x_1,1),(x_1,x_2))$*, meaning this fault cannot occur when* x_1 *outputs* 1.

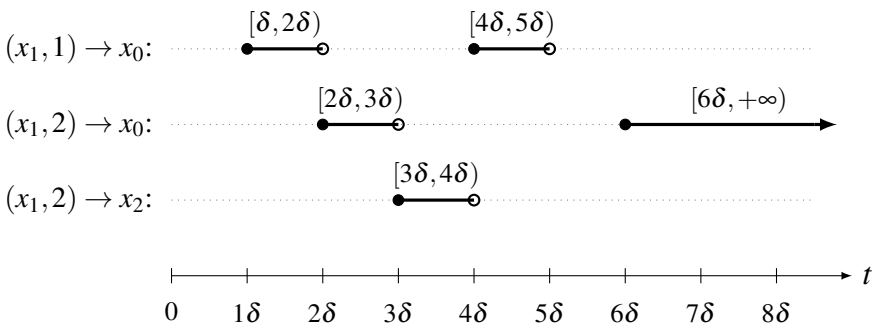


Figure 5.2: Fault enabling windows for Example 5.2.

5.2.3 Switching Output Automaton with Faults

We now formally define a Switching Output Automaton with Faults (SOAF) as an extension of the SOA, building on fault-prone states, fault arcs, and the fault time mapping.

5.2.3.1 Formal Definition

Definition 5.6 (Switching Output Automaton with Faults (SOAF))**.** *A* Switching Output Automaton with Faults *(SOAF) is the tuple*

$$G_f = \langle G, B_f, Q_f, \mathscr{I} \rangle,$$

where $G = (X, Y, B, h, x_0, y_0)$ *is an SOA;* $B_f \subseteq X_f \times X$ *is the set of fault arcs with* $X_f \subseteq X$ *the fault-prone states;* Q_f *is the set of fault-prone global states (Definition 5.4); and* $\mathscr{I}$ *is the fault time mapping (Definition 5.5).*

5.2.3.2 Semantics and Evolution

The SOAF extends the SOA semantics from Chapter 4 by introducing an additional layer of nondeterministic fault transitions. While the nominal behavior follows the same minimal dwell time constraint δ, fault transitions are further restricted by time windows defined through the fault time mapping $\mathscr{I}$.

Event atomicity and enablement. At any dwell time t, at most one transition fires (either nominal or fault). When $t \geq \delta$ and some fault arc is time-enabled, both nominal and fault transitions may be concurrently available; the choice is nondeterministic.

SOAF augments the nominal SOA dynamics with time-gated fault transitions. While dwelling in state $x \in X$:

- **Nominal:** any $(x, x') \in B$ becomes enabled once the dwell time reaches δ.
- **Fault:** if $x \in X_f$ and the current global state is $q = (x, y)$, a fault arc $b_f = (x, x')$ is enabled only when the dwell time t lies in some interval $[\delta'_m, \delta''_m) \in \mathscr{I}(q, b_f)$; taking b_f is nondeterministic.

This nondeterminism models the inherent uncertainty of failures: satisfying the time gate makes a fault possible, not inevitable.

5.2.3.3 Observable Behavior

As in Chapter 4, only the output value and its dwell duration are observable; internal transitions are hidden. Thus outputs alone do not reveal whether a step is nominal (B) or faulty (B_f).

As in Chapter 4, we represent observations as sequences of timed output symbols (y, τ) where $y \in Y$ is an output value and $\tau \in \mathbb{R}_{\geq \delta}$ is the duration for which the output remains at value y.

Definition 5.7 (Language of a SOAF under minimal dwell time δ)**.** *Fix $\delta > 0$. The* language $L_\delta(G_f)$ *is the set of all observation sequences that can be generated by runs of G_f (including runs with fault occurrences):*

$$
\begin{aligned}
L_\delta(G_f) = \{\omega \in (Y \times \mathbb{R}_{\geq \delta})^+ \mid & \exists \text{ a run } \rho \text{ of } G_f \text{ that satisfies the minimal} \\
& \text{dwell-time constraint with } \delta, \text{ and a time partition} \\
& 0 = s_0 < s_1 < \cdots < s_{n+1} \text{ such that:} \\
& \omega = (y_0, \tau_0)(y_1, \tau_1) \cdots (y_n, \tau_n), \\
& y(t) = y_i \text{ for all } t \in [s_i, s_{i+1}), \\
& \tau_i = s_{i+1} - s_i \geq \delta, \\
& y_i \neq y_{i+1} \text{ for } i = 0, \ldots, n-1\}
\end{aligned} \tag{5.1}
$$

where $(Y \times \mathbb{R}_{\geq \delta})^+$ denotes the set of all finite-length non-empty sequences of timed output symbols.

Remark 5.3. *The language $L_\delta(G_f)$ extends the nominal SOA language $L_\delta(G)$ in the following sense: $L_\delta(G) \subseteq L_\delta(G_f)$, where $L_\delta(G)$ contains all observation sequences generated exclusively through nominal arcs B, while $L_\delta(G_f)$ additionally includes sequences that involve fault arcs B_f. The key challenge in fault diagnosis is determining, for each observation sequence in $L_\delta(G_f)$, whether it can only be generated by nominal runs (certainly fault-free), only by faulty runs (certainly faulty), or by both (ambiguous). A comprehensive example illustrating these distinctions is presented in Chapter 6.*

5.2.3.4 Example: Complete SOAF Specification

We now present a complete example that integrates all the components of the SOAF framework.

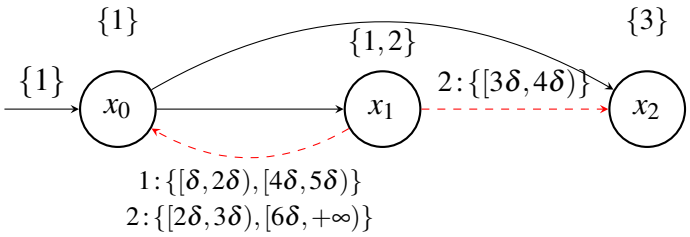


Figure 5.3: Complete SOAF example with initial state x_0. Nominal arcs (solid black) and fault arcs (dashed red) show fault time mappings conditioned on current output.

Example 5.3 (Complete SOAF)**.** *We construct a complete SOAF by combining the structural and temporal components introduced in previous sections, as illustrated in Figure 5.3.*

Base SOA. *Consider the SOA* $G = (X, Y, B, h, x_0, y_0)$ *with state set* $X = \{x_0, x_1, x_2\}$*, output alphabet* $Y = \{1, 2, 3\}$*, nominal arcs* $B = \{(x_0, x_1), (x_0, x_2)\}$*, and output function* $h(x_0) = \{1\}$*,* $h(x_1) = \{1, 2\}$*,* $h(x_2) = \{3\}$*. The system starts in initial configuration* $(x_0, 1)$*. State* x_1 *has no nominal outgoing transitions.*

Fault structure. *We extend the base SOA with fault arcs* $B_f = \{(x_1, x_0), (x_1, x_2)\}$ *that are disjoint from B. This yields fault-prone states* $X_f = \{x_1\}$ *and fault-prone global states* $Q_f = \{(x_1, 1), (x_1, 2)\}$.

Fault time mapping. *The temporal constraints specify when each fault can occur:*

- $\mathscr{I}((x_1, 1), (x_1, x_0)) = \{[\delta, 2\delta), [4\delta, 5\delta)\}$*: Fault to* x_0 *from* $(x_1, 1)$ *enabled in two disjoint windows.*
- $\mathscr{I}((x_1, 2), (x_1, x_0)) = \{[2\delta, 3\delta), [6\delta, +\infty)\}$*: Fault to* x_0 *from* $(x_1, 2)$ *enabled early and then persistently after* 6δ.
- $\mathscr{I}((x_1, 2), (x_1, x_2)) = \{[3\delta, 4\delta)\}$*: Fault to* x_2 *from* $(x_1, 2)$ *enabled only in mid-range.*

Note that $((x_1, 1), (x_1, x_2)) \notin D$*, meaning the fault to* x_2 *cannot occur when output is* 1.

Graphical notation. *In Figure 5.3, the labels on fault arcs use the notation "*$y : \{\ldots\}$*" to indicate the time intervals during which the fault can occur when the current output is* $y \in h(x_1)$*. For example, the fault arc* (x_1, x_0) *is labeled with both "*$1 : \{[\delta, 2\delta), [4\delta, 5\delta)\}$*" and "*$2 : \{[2\delta, 3\delta), [6\delta, +\infty)\}$*" because the fault windows depend on whether the system*

is currently producing output 1 *or* 2*. This output-conditioned representation captures the key insight that* $(x_1,1)$ *and* $(x_1,2)$ *admit different enabling intervals.*

Operational semantics. *From state* x_0*, the system can take nominal transitions to* x_1 *or* x_2 *after dwelling for at least* δ*. In the fault-prone state* x_1*, since there are no nominal outgoing arcs, the system can only evolve through fault transitions when they are time-enabled, or continue dwelling in* x_1 *otherwise. The specific fault possibilities depend on both the current output and elapsed dwell time: when output is* 1*, only the fault to* x_0 *is possible during* $[\delta,2\delta)$ *or* $[4\delta,5\delta)$*; when output is* 2*, the fault to* x_0 *is possible during* $[2\delta,3\delta)$ *or after* 6δ*, and the fault to* x_2 *becomes possible during* $[3\delta,4\delta)$*. State* x_2 *is absorbing with no outgoing transitions.*

5.3 Fault Diagnosis Framework

Compared to classical DES diagnosis [10], SOAF diagnosis must handle timed output observations and time-windowed fault enablement, under partial observability and set-valued outputs (see Sections 5.1–5.2.3). We address these challenges through a four-stage pipeline that transforms a SOAF into a diagnoser capable of online fault classification while preserving the essential timing semantics.

Pipeline architecture. Given a SOAF $G_f = \langle G, B_f, Q_f, \mathscr{I} \rangle$ with fault windows $\mathscr{I}$ and minimal dwell time $\delta > 0$, our objective is to construct a deterministic diagnoser G_{diag} for online fault classification. The pipeline constructs the following components in sequence:

$$G_f \xrightarrow{\text{Stages 1+2}} G_{ef} \xrightarrow{\text{Stage 3}} \text{Rec}(G_{ef}) \xrightarrow{\text{Stage 4}} G_{diag}$$

where G_{ef} is the evolution automaton with faults (built on the logical state space Q_e constructed in Stage 1), $\text{Rec}(G_{ef})$ is the fault recognizer, and G_{diag} is the deterministic diagnoser.

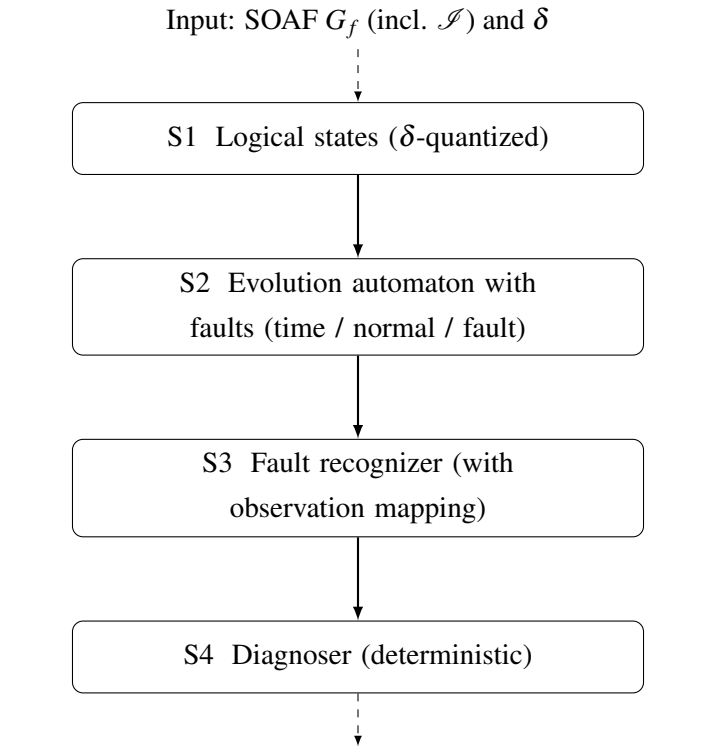


Figure 5.4: Four-stage diagnosis pipeline: S1 logical states; S2 evolution automaton with faults; S3 fault recognizer with observation mapping; S4 deterministic diagnoser.

The construction proceeds through four stages (Figure 5.4):

Stage 1 — Logical global states (Section 5.3.1). For each global state, discretize the continuous dwell time into intervals of length δ, creating logical states $(x,y)_j$ that encode the elapsed dwell time. This yields a finite state space while preserving fault-window boundaries.

Stage 2 — Evolution Automaton with Faults (Section 5.3.2). On this set, build an automaton with three kinds of moves: time elapse, normal transitions, and fault transitions enabled only within specified time windows. This enumerates all admissible evolutions.

Stage 3 — Fault recognizer (Section 5.3.3). Compose the automaton with a simple two-state monitor (normal/faulty) and add an observation mapping to mark which moves are visible to an external observer.

Stage 4 — Diagnoser (Section 5.3.4). Determinize the recognizer to maintain the set of all situations compatible with the observed outputs, and classify the current situation as normal, faulty, or uncertain for online use.

Remark 5.4 (Relationship to Existing Frameworks)**.** *This pipeline generalizes the Sampath–Lafortune diagnoser [10] from DES to the timed SOAF setting. The novelty lies in Stage 1's discretization tailored to fault windows, which preserves time-window semantics while enabling a finite abstraction for diagnosis.*

5.3.1 Construction of Logical Global States

The construction of logical states for fault diagnosis presents a fundamentally different challenge compared to the opacity verification in Chapter 4. The key distinction lies not merely in the number of time intervals to track, but in the *structural complexity* of what must be distinguished. While opacity verification determines whether the system is within a secret dwell pattern (a property of a single state), fault diagnosis must *simultaneously track multiple distinct fault mechanisms*, each associated with a different fault arc originating from the same state. Specifically:

- **Opacity (Chapter 4):** For each vulnerable global state $q \in Q_v$, there is one secret pattern $\mathscr{I}_v(q)$ (which may consist of one or more intervals). The goal is to hide whether the system is "in secret" or "not in secret"—a *binary classification* per global state.
- **Fault Diagnosis (this chapter):** For each fault-prone global state $q \in Q_f$, there may be multiple outgoing fault arcs, each with its own timing specification (which may also consist of multiple intervals). The goal is to maintain the *per-arc enablement status* and, upon observing a fault, distinguish the possible failure modes based on timing and output information. This requires tracking enablement information for each individual fault arc, rather than a single binary label per state.

Consider a fault-prone global state $q = (x,y)$ where: dwelling for $[2\delta, 3\delta)$ may cause electrical failure via one fault arc (leading to emergency shutdown); dwelling for $[3\delta, 4\delta)$ may cause thermal failure via a different fault arc (leading to thermal alarm); and dwelling for $[6\delta, +\infty)$ may again enable the first fault arc. This intricate temporal structure—where different fault arcs may have multiple, possibly overlapping activation windows—necessitates tracking the fault enablement status for *each individual arc* at every logical state, a requirement absent in opacity verification.

To address this challenge, we partition the continuous time dimension into intervals that preserve all fault-window boundaries. A *logical global state* $(x,y)_j$ represents the configuration where the system is in global state (x,y) with dwell time in the j-th interval I_j (defined below). The granularity of this discretization is determined by the *depth*

function $\mathscr{R} : Q \to \mathbb{N}_+$ (analogous to the depth function in Chapter 4, but adapted for fault diagnosis; see formal definition in (5.2) below), which computes the maximum logical state index needed for each global state based on all fault-window boundaries:

Non–fault-prone global states. If $q = (x,y) \notin Q_f$, no fault can occur while dwelling in q. We set $\mathscr{R}(q) = 1$. The intervals of interest are:

$$I_0 = [0, \delta) \quad \text{and} \quad I_1 = [\delta, +\infty)$$

corresponding to the logical state sequence

$$(x,y)_0 \xrightarrow{\delta} (x,y)_1 \circlearrowleft_{\delta}$$

This creates two logical states: $(x,y)_0$ represents the initial interval before the minimal dwell time has elapsed, and $(x,y)_1$ represents all subsequent time with a self-loop for indefinite dwelling.

Fault-prone global states. The discretization of fault-prone states $q = (x,y) \in Q_f$ requires careful coordination of multiple fault windows. Let $B_{f,x} = \{b^1_{f,x}, \ldots, b^r_{f,x}\}$ denote the r fault arcs originating from discrete state x (where $r = |B_{f,x}|$ is the number of outgoing fault arcs). Each arc $b^s_{f,x}$ (for $s \in \{1, \ldots, r\}$) has an associated set of fault windows:

$$\mathscr{I}(q, b^s_{f,x}) = \{[k'_{1,s}\delta, k''_{1,s}\delta), \ldots, [k'_{n_s,s}\delta, k''_{n_s,s}\delta)\}$$

where n_s denotes the number of fault windows for arc $b^s_{f,x}$, and the boundary indices $k'_{i,s} \in \mathbb{N}_+$ and $k''_{i,s} \in \mathbb{N}_+ \cup \{+\infty\}$ specify the left and right endpoints of the i-th window (for $i \in \{1, \ldots, n_s\}$).

These windows may exhibit complex relationships: windows from different arcs may overlap (creating periods of multiple concurrent vulnerabilities), nest within each other (where one failure mode is a special case of another), or appear disjoint (representing independent failure mechanisms). To preserve all temporal distinctions necessary for diagnosis, we employ a *boundary aggregation algorithm*:

Step 1: Boundary collection. We aggregate all finite time boundaries from all fault arcs:

$$\mathscr{B}(q) = \{1\} \cup \{k \in \mathbb{N}_+ \mid \exists s \in [1,r], \exists i \in [1, n_s] : k \in \{k'_{i,s}, k''_{i,s}\}\}$$

This set contains all critical time points where fault enablement status changes. The inclusion of $\{1\}$ ensures that the minimal dwell time boundary δ (i.e., $1 \times \delta$) is always

represented, while the condition $k \in \mathbb{N}_+$ filters out the unbounded right endpoint $+\infty$ (when $k''_{i,s} = +\infty$).

Step 2: Maximum index determination. The depth function $\mathscr{R}(q)$ is computed as the maximum boundary index:

$$\mathscr{R}(q) = \max \mathscr{B}(q) \tag{5.2}$$

Note that $\mathscr{B}(q) \neq \emptyset$ is always guaranteed since $1 \in \mathscr{B}(q)$ by construction in Step 1.

For a fault-prone state with $\mathscr{R}(q) = k$, the intervals of interest are:

$$I_0 = [0, \delta), I_1 = [\delta, 2\delta), \ldots, I_{k-1} = [(k-1)\delta, k\delta), I_k = [k\delta, +\infty)$$

corresponding to the logical state sequence

$$(x,y)_0 \xrightarrow{\delta} (x,y)_1 \xrightarrow{\delta} \ldots \xrightarrow{\delta} (x,y)_{k-1} \xrightarrow{\delta} (x,y)_k \circlearrowleft_{\delta}$$

This creates $k+1$ logical states, with δ-transitions between consecutive states and a self-loop on the final state $(x,y)_k$.

Definition 5.8 (Fault Enablement Function)**.** *The* fault enablement function $E : Q_f \times \mathbb{N} \times B_f \to \{true, false\}$ *determines when each fault arc becomes active at each logical state. For a fault-prone global state $q \in Q_f$ with logical state index $j \in \{0, 1, \ldots, \mathscr{R}(q)\}$ and fault arc $b^s_{f,x} \in B_f$:*

$$E(q, j, b^s_{f,x}) = \begin{cases} true, & \text{if } I_j \cap \bigcup_{i=1}^{n_s} [k'_{i,s}\delta, k''_{i,s}\delta) \neq \emptyset \\ false, & \text{otherwise} \end{cases}$$

where I_j denotes the j-th time interval for state q as defined by the boundary aggregation algorithm, and n_s denotes the number of fault windows for arc $b^s_{f,x}$.

This function encodes the complex temporal logic of fault activation. A logical state $(x,y)_j$ may be concurrently enabled for multiple fault arcs (concurrent vulnerabilities), enabled for some arcs but not others (selective vulnerability), or enabled for no arcs (safe dwelling period). When $k''_{i,s} = +\infty$, the final logical state $(x,y)_{\mathscr{R}(q)}$ remains permanently enabled for that fault arc, modeling persistent vulnerability under prolonged stress.

Semantic properties. The logical state construction enforces the following behavioral constraints:

- **Dwell-time enforcement:** The initial logical state $(x,y)_0$ enforces the minimal dwell time constraint—no transitions (normal or faulty) can occur until δ time units elapse.
- **Transition availability:** For $j > 0$, logical state $(x,y)_j$ permits normal transitions according to the behavior relation B and output function h, as well as fault transitions along arcs $b^s_{f,x}$ satisfying $E(q,j,b^s_{f,x}) = \text{true}$.
- **Diagnostic information preservation:** Unlike opacity where we only track whether the system is "in secret" or "not in secret", the fault enablement structure maintains *which specific faults* are possible at each logical state, providing the granular information necessary for differential diagnosis.

The following table summarizes the key structural differences:

Global State Type	$\mathscr{R}(q)$	**# Logical States**	**Fault-Enabled Indices**
Non–fault-prone $(q \notin Q_f)$	1	2	None
Fault-prone $(q \in Q_f)$	$\max \mathscr{B}(q)$	$\mathscr{R}(q)+1$	Determined by $E(q,j,b^s_{f,x})$

Table 5.1: Logical state discretization for fault-prone and non-fault-prone global states.

Example 5.4 (Logical States for Fault-Prone System)**.** *Consider the SOAF from Example 5.3. We compute the logical states for each global state:*

Non–fault-prone global states:

- $q = (x_0,1)$*: Since* $h(x_0) = \{1\}$*, this is the only valid global state for* x_0*. It has* $\mathscr{R}(q) = 1$*, yielding two logical states* $(x_0,1)_0$ *and* $(x_0,1)_1$*.*
- $q = (x_2,3)$*: Since* $h(x_2) = \{3\}$*, this is the only valid global state for* x_2*. It has* $\mathscr{R}(q) = 1$*, yielding two logical states* $(x_2,3)_0$ *and* $(x_2,3)_1$*.*

Fault-prone global states:

- $q_1 = (x_1,1) \in Q_f$ *with fault windows* $\mathscr{I}(q_1,(x_1,x_0)) = \{[\delta,2\delta),[4\delta,5\delta)\}$*:*
  - *Boundary collection:* $\mathscr{B}(q_1) = \{1,2,4,5\}$
  - *Depth function:* $\mathscr{R}(q_1) = 5$
  - *Logical states:* $(x_1,1)_0,(x_1,1)_1,\ldots,(x_1,1)_5$ *(6 states total)*
  - *Fault-enabled indices for* (x_1,x_0)*:* $j \in \{1,4\}$
- $q_2 = (x_1,2) \in Q_f$ *with two different fault arcs:*

- $\mathscr{I}(q_2,(x_1,x_0)) = \{[2\delta,3\delta),[6\delta,+\infty)\}$
- $\mathscr{I}(q_2,(x_1,x_2)) = \{[3\delta,4\delta)\}$
- *Boundary collection:* $\mathscr{B}(q_2) = \{1,2,3,4,6\}$
- *Depth function:* $\mathscr{R}(q_2) = 6$
- *Logical states:* $(x_1,2)_0,(x_1,2)_1,\ldots,(x_1,2)_6$ *(7 states total)*
- *Fault-enabled indices: - For* (x_1,x_0)*:* $j \in \{2,6\}$ *(note:* $j = 6$ *captures the* $[6\delta,+\infty)$ *window) - For* (x_1,x_2)*:* $j \in \{3\}$

The time evolution for $q_2 = (x_1,2)$ *demonstrates multiple fault windows:*

$$(x_1,2)_0 \xrightarrow{\delta} (x_1,2)_1 \xrightarrow{\delta} \underbrace{(x_1,2)_2}_{\text{can fault to } x_0} \xrightarrow{\delta} \underbrace{(x_1,2)_3}_{\text{can fault to } x_2}$$
$$\xrightarrow{\delta} (x_1,2)_4 \xrightarrow{\delta} (x_1,2)_5 \xrightarrow{\delta} \underbrace{(x_1,2)_6}_{\text{can fault to } x_0} \circlearrowleft_{\delta}$$

where different fault arcs become enabled at different dwell times, reflecting diverse failure mechanisms.

Proposition 5.1 (Partition, Finiteness, and Fault-Window Preservation)**.** *Let* $\delta > 0$ *be the minimal dwell time. For each global state* $q = (x,y) \in Q$*, let* $\mathscr{R}(q) \in \mathbb{N}_{\geq 1}$ *be the depth function defined above, and*

$$I_j = \begin{cases} [j\delta,(j+1)\delta), & 0 \leq j < \mathscr{R}(q), \\ [\mathscr{R}(q)\delta,+\infty), & j = \mathscr{R}(q). \end{cases}$$

We interpret $t \geq 0$ *as the dwell time spent in* q *since the entry into* q*. Define the set of logical states by*

$$Q_e = \{(x,y)_j \mid q = (x,y) \in Q,\ 0 \leq j \leq \mathscr{R}(q)\}.$$

For $q \in Q_f$ *and a fault arc* $b^s_{f,x} \in B_f$ *originating from* x*, let* n_s *denote the number of fault windows for arc* $b^s_{f,x}$*, with windows* $\{[k'_{i,s}\delta,k''_{i,s}\delta)\}_{i=1}^{n_s}$*. The enablement function* $E(q,j,b^s_{f,x})$ *is as defined previously.*

Then the logical state construction satisfies:

1. ***Partition.*** *For every* $t \geq 0$ *there exists a unique index*

$$j(t,q) = \min\left(\lfloor t/\delta \rfloor,\ \mathscr{R}(q)\right) \in \{0,\ldots,\mathscr{R}(q)\}$$

such that $t \in I_{j(t,q)}$*. Hence* $\{I_j\}_{j=0}^{\mathscr{R}(q)}$ *forms a partition of* $[0,+\infty)$*.*

2. ***Finiteness.*** *Q_e is finite and*

$$|Q_e| = \sum_{q \in Q} \big(\mathscr{R}(q) + 1\big).$$

3. ***Fault-window preservation.*** *For every $q \in Q_f$, $b^s_{f,x} \in B_f$ and $j \in \{0, \ldots, \mathscr{R}(q)\}$,*

$$E(q, j, b^s_{f,x}) = \mathit{true} \quad \Longleftrightarrow \quad \exists t \in I_j : \; t \in \bigcup_{i=1}^{n_s} [k'_{i,s}\delta, k''_{i,s}\delta).$$

In particular:

- (Soundness) *If the system executes $b^s_{f,x}$ at dwell time t from q, then with $j = j(t,q)$ we have $E(q, j, b^s_{f,x}) = \mathit{true}$.*
- (Completeness w.r.t. enablement) *If $E(q, j, b^s_{f,x}) = \mathit{true}$ with $j \geq 1$ (respecting the δ-dwell constraint), then there exists some $t \in I_j$ at which $b^s_{f,x}$ can be executed according to the SOAF semantics.*

Proof. The partition and finiteness properties follow directly from the definitions. For fault-window preservation, the equivalence holds because $E(q, j, b^s_{f,x}) = \text{true}$ if and only if I_j has non-empty intersection with some fault window, which is equivalent to the existence of $t \in I_j$ belonging to some fault window. The soundness and completeness properties follow from this equivalence and the SOAF semantics. □

5.3.2 Construction of the Evolution Automaton with Faults

The logical state construction in Section 5.3.1 provides a finite discretization of the continuous time domain. This discretization preserves fault-window boundaries while enabling state-based modeling. To perform diagnosis, we must now capture how the system evolves through these logical states under both normal operation and fault occurrences. This necessitates constructing an automaton that captures the complete dynamics of the SOAF.

The Evolution Automaton with Faults (EAF), denoted G_{ef}, serves this purpose by providing a finite-state model where each transition is explicitly labeled as either nominal (n) or faulty (f). This labeling is crucial: while observers see only output changes, the diagnoser must distinguish between intended behavior and faults. The EAF makes this distinction explicit in its structure, enabling systematic construction of the fault recognizer and ultimately the diagnoser.

5.3.2.1 Transition Semantics

The EAF transition structure must capture two essential aspects: the temporal and discrete dynamics of the SOAF, and the distinction between intended and faulty behavior. We partition the transition relation into three categories based on their triggering mechanisms and observability properties.

Time-driven transitions. Time-driven transitions capture the autonomous passage of time within a global state as the system dwells. For a logical state $(x,y)_j$, time progression is modeled as:

$$(x,y)_j \xrightarrow{n} \begin{cases} (x,y)_{j+1} & \text{if } j < \mathscr{R}(x,y) \\ (x,y)_j & \text{if } j = \mathscr{R}(x,y) \text{ (self-loop)} \end{cases} \tag{5.3}$$

These transitions do not change the output—only internal timers progress—and are labeled n in the EAF (see Remark 5.5 for the distinction between internal event labels and observation symbols). Although output-preserving, the passage of time itself is physically observable through timing measurements, represented by the logical observation symbol δ (Section 5.3.3). These transitions are crucial for diagnosis as they determine precisely when fault windows open and close. The self-loop at $j = \mathscr{R}(x,y)$ represents states that have exhausted all critical time boundaries, allowing indefinite dwelling without further time discretization.

Nominal transitions. Nominal transitions capture the intended system behavior by executing arcs in B. To enforce the δ-dwell time constraint, these transitions are enabled only when $j > 0$, ensuring the system has dwelt for at least one time interval:

$$(x,y)_j \xrightarrow{n} (\bar{x},\bar{y})_0 \quad \text{where } j > 0, (\bar{x},\bar{y}) \in \Gamma((x,y)) \tag{5.4}$$

The target states are defined by $\Gamma((x,y)) = \{(\bar{x},\bar{y}) \mid \bar{x} \in \sigma(x), \bar{y} \in h(\bar{x})\}$, representing all global states reachable via nominal arcs. The nondeterminism in Γ arises from two sources:

- **Structural nondeterminism**: Multiple outgoing arcs from state x, creating choice among successor states in $\sigma(x)$
- **Output nondeterminism**: Multiple possible outputs at the target state $\bar{x}$, creating ambiguity in $h(\bar{x})$

This nondeterminism creates a fundamental diagnostic challenge: a single observable output change may correspond to multiple underlying state transitions, making it impossible

to determine the exact system state from observations alone. This requires the diagnoser to maintain belief states rather than tracking a single deterministic state.

Fault transitions. Fault transitions model deviations from nominal behavior triggered by fault occurrences. For a fault arc $b^s_{f,x} = (x, x') \in B_f$ from state x to state x', unlike nominal transitions that become enabled after any sufficient dwell, fault transitions are time-gated by fault windows:

$$(x,y)_j \xrightarrow{f} (x',y')_0 \quad \text{if and only if } E((x,y), j, (x,x')) = \text{true} \tag{5.5}$$

The enablement function E (Definition 5.8) returns true if and only if the current dwell time interval I_j intersects the fault window for arc (x,x'). The explicit fault label f distinguishes these transitions from nominal behavior, which is essential for diagnosing system health. This introduces two critical forms of nondeterminism beyond those in nominal behavior: multiple fault arcs may be concurrently enabled at a logical state, and fault transitions compete with nominal transitions for execution. Both forms complicate the diagnostic inference problem.

Observability and diagnostic implications. The EAF exhibits partial observability through two types of observable events: output changes ($y \to y'$) and time progression (represented by the logical symbol δ). However, non-time-driven output-preserving transitions remain hidden from observers, creating fundamental diagnostic challenges. This creates two key complications:

1. **Detection ambiguity**: An observed output change $y \to y'$ may result from either a nominal transition or a fault transition. Even with timing information, multiple explanations may remain consistent with the observation.

2. **Uncertainty accumulation**: Non-time-driven output-preserving transitions can silently alter the system state and health status without producing any observable event. Faults may remain undetected until a subsequent output change or time progression reveals the deviation.

These complications require the diagnoser to maintain *belief states*: sets of possible logical states paired with their health status (nominal or faulty), consistent with the observed sequence of output changes and time progressions, rather than tracking a single deterministic state.

5.3.2.2 Formal Definition

Definition 5.9 (Evolution Automaton with Faults)**.** *Given a SOAF* $G_f = \langle G, B_f, Q_f, \mathscr{I} \rangle$ *where* $G = (X, Y, B, h, \sigma, x_0, y_0)$ *with global state set* $Q = X \times Y$*, its* Evolution Automaton with Faults *(EAF) is a nondeterministic finite automaton* $G_{ef} = (Q_e, \Sigma, \Delta, q_0)$ *where:*

- $Q_e = \{(x,y)_j \mid (x,y) \in Q, 0 \leq j \leq \mathscr{R}(x,y)\}$ *is the finite set of logical states, where* $\mathscr{R}$ *is the depth function (Definition 5.2);*
- $\Sigma = \{n, f\}$ *is the alphabet (n: nominal, f: fault);*
- $\Delta = \Delta_{time} \cup \Delta_{nom} \cup \Delta_{fault} \subseteq Q_e \times \Sigma \times Q_e$ *is the transition relation, where* E *is the enablement function (Definition 5.8), partitioned as:*

$$\Delta_{time} = \{((x,y)_j, n, (x,y)_{j+1}) \mid j < \mathscr{R}(x,y)\} \cup \{((x,y)_j, n, (x,y)_j) \mid j = \mathscr{R}(x,y)\}$$
$$\Delta_{nom} = \{((x,y)_j, n, (\bar{x},\bar{y})_0) \mid j > 0, (\bar{x},\bar{y}) \in \Gamma((x,y))\}$$
$$\Delta_{fault} = \{((x,y)_j, f, (x',y')_0) \mid (x,x') \in B_f, E((x,y), j, (x,x')) = true, y' \in h(x')\}$$

  where $\Gamma((x,y)) = \{(\bar{x},\bar{y}) \mid \bar{x} \in \sigma(x), \bar{y} \in h(\bar{x})\}$ *represents the set of global states reachable from* (x,y) *via nominal arcs.*

- $q_0 = (x_0, y_0)_0$ *is the initial state.*

Remark 5.5 (Nondeterminism and Observability in the EAF)**.** ***Competition-induced nondeterminism.*** *Multiple fault arcs may be concurrently enabled at a logical state* $(x,y)_j$*, and these coexist with enabled nominal transitions. The EAF preserves this nondeterminism, which is essential for capturing all possible system evolutions and enables the diagnoser to consider all potential explanations for observed behavior.*

Note. *The internal event alphabet of the EAF is* $\Sigma = \{n, f\}$*. The symbol* δ *belongs to the logical observation alphabet introduced later and represents physically observable time progression; it is not an internal event label of the EAF itself.*

The formal definition above specifies the structure of the EAF declaratively. To facilitate practical implementation and provide computational insights, Algorithm 6 presents an explicit construction procedure that systematically builds the logical state space Q_e and transition relation Δ from the components of the SOAF. The algorithm follows a three-phase structure: constructing logical states (Phase 1), generating transitions according to the semantics in Section 5.3.2.1 (Phase 2), and initializing the automaton (Phase 3).

Example 5.5 (EAF Construction)**.** *Consider the SOAF from Example 5.3. We construct its EAF step by step (a partial view is shown in Figure 5.5).*

Algorithm 6 Construction of Evolution Automaton with Faults

Input: SOAF $G_f = \langle G, B_f, Q_f, \mathscr{I} \rangle$ with $G = (X, Y, B, h, \sigma, x_0, y_0)$
Output: EAF $G_{ef} = (Q_e, \Sigma, \Delta, q_0)$
Phase 1: Construct logical state space
for each global state $(x,y) \in Q$ **do**
    Compute $\mathscr{R}(x,y)$ using Definition 5.2
    Add states $\{(x,y)_j \mid j \in \{0,1,\ldots,\mathscr{R}(x,y)\}\}$ to Q_e
end for
Phase 2: Construct transition relation
Initialize $\Delta \leftarrow \emptyset$
for each $(x,y)_j \in Q_e$ **do**
    // Time-driven transitions
    if $j < \mathscr{R}(x,y)$ **then**
        Add $((x,y)_j, n, (x,y)_{j+1})$ to Δ
    else
        Add $((x,y)_j, n, (x,y)_j)$ to Δ **// self-loop**
    end if
    // Nominal transitions
    if $j > 0$ **then**
        for each $\bar{x} \in \sigma(x)$ **do**
            for each $\bar{y} \in h(\bar{x})$ **do**
                Add $((x,y)_j, n, (\bar{x},\bar{y})_0)$ to Δ
            end for
        end for
    end if
    // Fault transitions
    for each x' such that $(x,x') \in B_f$ **do**
        if $E((x,y), j, (x,x')) = \text{true}$ **then**
            for each $y' \in h(x')$ **do**
                Add $((x,y)_j, f, (x',y')_0)$ to Δ
            end for
        end if
    end for
end for
Phase 3: Set initial state and alphabet
$q_0 \leftarrow (x_0,y_0)_0$
$\Sigma \leftarrow \{n, f\}$
return $G_{ef} = (Q_e, \Sigma, \Delta, q_0)$

Step 1: Logical state space. *For non–fault-prone states,* $\mathscr{R}(x_0,1) = \mathscr{R}(x_2,3) = 1$*, yielding logical states:*

- $(x_0,1)_0, (x_0,1)_1$
- $(x_2,3)_0, (x_2,3)_1$

For fault-prone states in $Q_f = \{(x_1,1),(x_1,2)\}$*:*

- *For* $(x_1,1)$*: fault windows end at* 5δ*, so* $\mathscr{R}(x_1,1) = 5$
- *For* $(x_1,2)$*: fault windows include* $[6\delta,+\infty)$*, so* $\mathscr{R}(x_1,2) = 6$

This yields logical states $(x_1,1)_j$ *for* $j \in \{0,1,\ldots,5\}$ *and* $(x_1,2)_j$ *for* $j \in \{0,1,\ldots,6\}$.

Step 2: Fault enablement. *Using the fault time mapping from Example 5.3:*

- $E((x_1,1),j,(x_1,x_0)) =$ *true for* $j \in \{1,4\}$ *(intervals* $[\delta,2\delta)$ *and* $[4\delta,5\delta)$*)*
- $E((x_1,2),j,(x_1,x_0)) =$ *true for* $j \in \{2,6\}$ *(intervals* $[2\delta,3\delta)$ *and* $[6\delta,+\infty)$*)*
- $E((x_1,2),j,(x_1,x_2)) =$ *true for* $j = 3$ *(interval* $[3\delta,4\delta)$*)*

The resulting EAF contains 17 logical states with transitions labeled n (time-driven and nominal) and f (fault), as specified in Definition 5.9.

Remark 5.6 (Complexity Analysis)**.** *The EAF has* $|Q_e| = O(|X| \cdot |Y| \cdot k_{\max})$ *states where* $k_{\max} = \max_{q \in Q} \mathscr{R}(q)$ *is the maximum depth across all global states. Here Y denotes the output alphabet of the original SOAF (distinct from the event alphabet* $\Sigma = \{n,f\}$ *of the EAF).*

The transition relation size can be decomposed as:

- $|\Delta_{time}| = |Q_e|$ *(exactly one time transition per logical state)*
- $|\Delta_{nom}| \leq \sum_{(x,y) \in Q} \mathscr{R}(x,y) \cdot |\sigma(x)| \cdot |Y|$
- $|\Delta_{fault}| \leq \sum_{(x,y) \in Q_f} \mathscr{R}(x,y) \cdot |B_f(x)| \cdot |Y|$

where $|\sigma(x)|$ *denotes the number of successor states reachable from x via nominal arcs, and* $|B_f(x)| = |\{x' \mid (x,x') \in B_f\}|$ *denotes the number of outgoing fault arcs from state x.*

Despite this potential state explosion, the EAF remains tractable for practical systems due to:

1. *The sparsity of fault-prone states (typically* $|X_f| \ll |X|$*)*

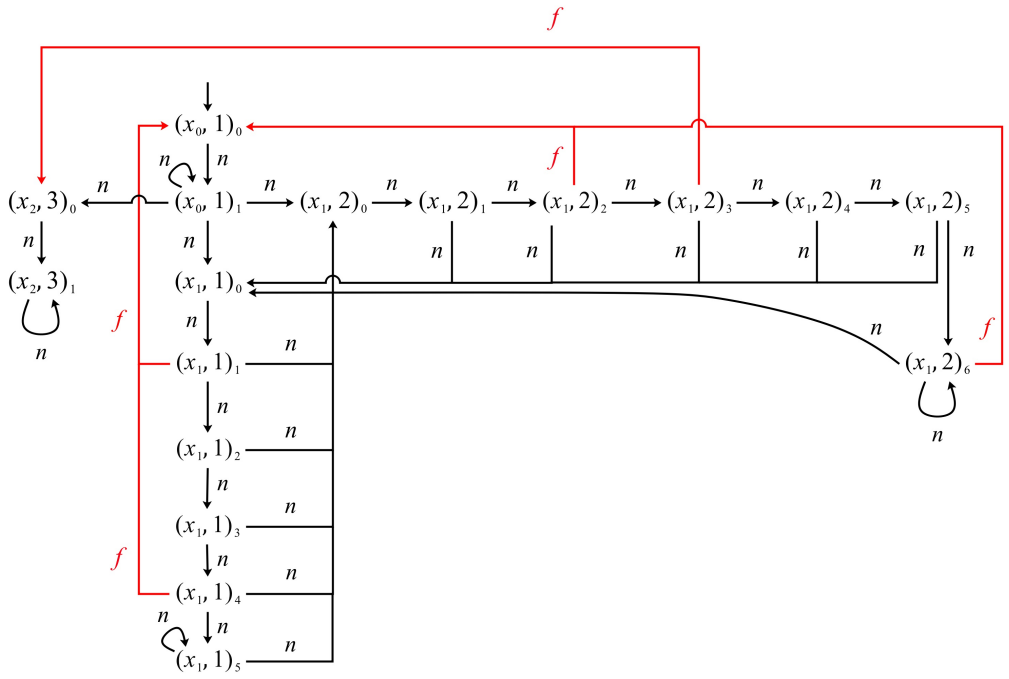


Figure 5.5: Evolution Automaton with Faults for Example 5.5.

2. *The bounded nature of $k_{\max}$ (fault windows rarely extend beyond a few multiples of δ)*

3. *The structural constraints imposed by the output function h*

In practice, industrial systems with 10-20 discrete states and fault windows extending to 10δ yield EAFs with hundreds to thousands of states—well within the capabilities of modern diagnostic algorithms. Furthermore, symbolic representations and on-the-fly construction techniques can handle significantly larger systems without explicitly enumerating the entire state space.

5.3.2.3 Language Preservation

The EAF construction preserves the essential semantics of the original SOAF at the granularity of the minimal dwell time δ. Since the EAF advances time in steps of size δ, its projected timed output language corresponds to the δ-quantized version of the SOAF language. We make this precise below. Note that the projection π maps logical states to the output alphabet Y (not the event alphabet $\Sigma = \{n, f\}$), allowing us to compare the externally observable behaviors of the SOAF and its EAF abstraction.

Definition 5.10 (δ-Quantization of Timed Outputs)**.** *For a timed output sequence*

$$\omega = (y_0, \tau_0)(y_1, \tau_1) \cdots (y_n, \tau_n)$$

with $\tau_i \geq \delta$, define the δ-quantization $Q_\delta(\omega)$ by replacing each duration with the largest multiple of δ not exceeding it:

$$Q_\delta(\omega) = (y_0, k_0\delta)(y_1, k_1\delta) \cdots (y_n, k_n\delta), \quad k_i = \left\lfloor \tfrac{\tau_i}{\delta} \right\rfloor \geq 1.$$

For a language $L \subseteq (Y \times \mathbb{R}_{\geq\delta})^+$, define $Q_\delta(L) = \{Q_\delta(\omega) \mid \omega \in L\}$.

Theorem 5.1 (Language Preservation of EAF)**.** *Let G_f be a SOAF and G_{ef} be its corresponding EAF constructed by Algorithm 6. Define the output projection $\pi : Q_e \to Y$ as $\pi((x,y)_j) = y$. For each run in G_{ef}, the projected timed output sequence is obtained by (i) grouping consecutive logical states with identical outputs, and (ii) computing the accumulated dwell time for each output as the number of time-driven transitions (including self-loops) multiplied by δ. Then:*

$$\mathscr{L}_\pi(G_{ef}) = Q_\delta\big(L_\delta(G_f)\big)$$

where $\mathscr{L}_\pi(G_{ef})$ denotes the set of timed output sequences observable from runs of the EAF and Q_δ is as in Definition 5.10.

Proof. We establish the equivalence by showing mutual inclusion.

($\subseteq$) Let $\omega = (y_0, \tau_0)(y_1, \tau_1) \cdots (y_n, \tau_n)$ be a timed output sequence generated by a run ρ_{ef} of the EAF. We construct a corresponding run ρ_f of the SOAF whose δ-quantized observation equals ω:

1. Each sequence of k time-driven transitions while the output is y corresponds to dwelling in the global state (x,y) for cumulative duration $k\delta$ in the SOAF. If the index reaches $j = \mathscr{R}(x,y)$, additional time-driven transitions are realized via the self-loop at $j = \mathscr{R}(x,y)$.

2. Each nominal transition $(x,y)_j \xrightarrow{n} (\bar{x},\bar{y})_0$ with $j > 0$ corresponds to taking arc $(x,\bar{x}) \in B$ after dwelling in (x,y) for some time t satisfying: if $j < \mathscr{R}(x,y)$, then $t \in [j\delta, (j+1)\delta)$; if $j = \mathscr{R}(x,y)$, then $t \geq j\delta$. The exact choice of t does not affect Q_δ.

3. Each fault transition $(x,y)_j \xrightarrow{f} (x',y')_0$ with $E((x,y), j, (x,x')) = \text{true}$ corresponds to taking fault arc $(x,x') \in B_f$ at some time $t \in I_j \cap \bigcup \mathscr{I}((x,y),(x,x'))$ (non-empty by enablement). If $j < \mathscr{R}(x,y)$, then $I_j = [j\delta, (j+1)\delta)$ and $\lfloor t/\delta \rfloor = j$; if $j = \mathscr{R}(x,y)$,

then $I_j = [j\delta, +\infty)$ and we may choose any $t \in I_j \cap \mathscr{I}((x,y),(x,x'))$ with $\lfloor t/\delta \rfloor = k$ for the appropriate $k \geq j$ matching the number of time-driven transitions in ρ_{ef}. In either case, the δ-quantized dwell time matches that produced by the EAF run.

By construction, the δ-quantized dwell times in ρ_f match those in ρ_{ef}, and therefore $Q_\delta(\text{obs}(\rho_f)) = \omega$.

($\supseteq$) Conversely, let $\omega = (y_0,\tau_0)(y_1,\tau_1)\cdots(y_n,\tau_n)$ be generated by a run ρ_f of the SOAF. Since SOAF enforces minimal dwell time δ, we have $\tau_i \geq \delta$ for all i. Set $k_i = \left\lfloor \frac{\tau_i}{\delta} \right\rfloor \geq 1$ and construct a corresponding run ρ_{ef} of the EAF that produces $Q_\delta(\omega)$:

1. For each dwell period in state (x,y) with duration τ_i, include k_i time-driven transitions while the output is y. If $k_i \leq \mathscr{R}(x,y)$, this reaches $(x,y)_{k_i}$; otherwise it reaches $(x,y)_{\mathscr{R}(x,y)}$ and accrues the remaining $k_i - \mathscr{R}(x,y)$ steps on the self-loop at $j = \mathscr{R}(x,y)$.

2. For each nominal transition from (x,y) to $(\bar{x},\bar{y})$ at the end of the dwell, include $(x,y)_j \xrightarrow{n} (\bar{x},\bar{y})_0$ with $j = \min\{k_i, \mathscr{R}(x,y)\}$. Since $k_i \geq 1$ and $\mathscr{R}(x,y) \geq 1$ by construction, we have $j = \min\{k_i, \mathscr{R}(x,y)\} \geq 1 > 0$, satisfying the nominal transition enablement condition.

3. For each fault transition from (x,y) to (x',y') occurring at time $t \in \mathscr{I}((x,y),(x,x'))$, let $j^* = \min\{\lfloor t/\delta \rfloor,\ \mathscr{R}(x,y)\}$. Since $t \in I_{\lfloor t/\delta \rfloor}$, we have $E((x,y), j^*, (x,x')) = \text{true}$ by Proposition 5.1. Include the transition $(x,y)_{j^*} \xrightarrow{f} (x',y')_0$.

The projection of ρ_{ef} yields the δ-quantized timed output sequence $Q_\delta(\omega)$, completing the proof. □

Remark 5.7 (Soundness for Diagnosis)**.** *The language preservation theorem establishes that the EAF faithfully captures both the nominal and faulty behaviors of the SOAF at the granularity of δ-quantization, making it a sound basis for fault diagnosis. Every observable behavior of the original SOAF is preserved in the EAF up to δ-discretization: behaviors differing only in continuous dwell times within a δ-interval are merged into a single discrete representative. This ensures that no diagnostic information is lost beyond the inherent time quantization, which is necessary for obtaining a finite-state abstraction.*

5.3.3 Construction of the Fault Recognizer

The EAF provides a complete enumeration of all possible system evolutions, both nominal and faulty. However, it lacks two essential features for online diagnosis: (i) explicit

tracking of whether a fault has occurred, and (ii) a mapping from transitions to observable outputs. We address these limitations by constructing a *fault recognizer* through the concurrent composition [8] of the EAF with a fault monitor. This recognizer forms the basis for the deterministic diagnoser developed in Section 5.3.4.

To maintain clarity throughout this section, we use $\Sigma = \{n, f\}$ as the event alphabet of the EAF (where n denotes nominal and f denotes fault events) and $\mathscr{O} = Y \cup \{\delta\}$ as the observation alphabet (where Y is the output alphabet of the SOAF and δ represents the passage of minimal dwell time).

5.3.3.1 Fault Monitor

To address the first limitation (explicit fault tracking), we introduce a simple two-state automaton that monitors the event labels in the EAF and maintains a binary health status: normal or faulty.

Definition 5.11 (Fault Monitor)**.** *Given an EAF $G_{ef} = (Q_e, \Sigma, \Delta, q_0)$ with event alphabet $\Sigma = \{n, f\}$, the* fault monitor *is a deterministic finite automaton $M = (X_M, \Sigma, \delta_M, x_{M,0})$ where:*

- *$X_M = \{N, F\}$ is the state set (N: normal, F: faulty)*
- *$x_{M,0} = N$ is the initial state (system starts in normal condition)*
- *$\delta_M : X_M \times \Sigma \to X_M$ is the transition function, where for $\gamma \in X_M$ and $a \in \Sigma$:*

$$\delta_M(\gamma, a) = \begin{cases} N & \text{if } \gamma = N \text{ and } a = n \text{ (remain normal)} \\ F & \text{if } \gamma = N \text{ and } a = f \text{ (fault occurs)} \\ F & \text{if } \gamma = F \text{ and } a \in \{n, f\} \text{ (stay faulty)} \end{cases} \tag{5.6}$$

The state F is absorbing: once a fault event f is observed, the monitor irreversibly transitions to and remains in state F, reflecting the permanent nature of fault occurrence in the diagnosis problem. The structure of the fault monitor is illustrated in Figure 5.6.

5.3.3.2 Fault Recognizer Construction

We now combine the EAF with the fault monitor to produce an automaton that explicitly tracks both the system's logical state and its health status. This is achieved through concurrent composition, where both components synchronize on every event: each transition in the EAF triggers a corresponding transition in the fault monitor, and the combined state space pairs EAF states with monitor states.

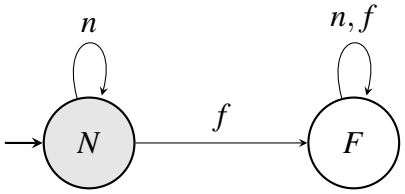


Figure 5.6: The fault monitor M.

Definition 5.12 (Fault Recognizer)**.** *Given an EAF $G_{ef} = (Q_e, \Sigma, \Delta, q_0)$ and fault monitor $M = (X_M, \Sigma, \delta_M, x_{M,0})$, the* fault recognizer *is the nondeterministic finite automaton obtained by their concurrent composition:*

$$Rec(G_{ef}) = G_{ef} \parallel M = (X_R, \Sigma, \Delta_R, x_{R,0})$$

where:

- ***State set:*** *$X_R = Q_e \times X_M = Q_e \times \{N, F\}$, pairing each EAF logical state with a fault label*
- ***Initial state:*** *$x_{R,0} = (q_0, N)$, starting in the initial EAF state with normal health status*
- ***Transition relation:*** *$\Delta_R = \{((q,\gamma), a, (q', \delta_M(\gamma,a))) \mid (q,a,q') \in \Delta, \gamma \in X_M\}$, where each EAF transition $(q,a,q') \in \Delta$ combined with each monitor state $\gamma \in \{N,F\}$ induces a recognizer transition that updates both the logical state ($q \to q'$) and the fault label ($\gamma \to \delta_M(\gamma,a)$) synchronously*

Observation mapping. The fault recognizer defined above still uses internal event labels $\Sigma = \{n, f\}$ for transitions, which are not directly observable. To enable online diagnosis based on observable outputs, we define an observation mapping $\mathscr{P} : \Delta_R \to \mathscr{O} \cup \{\varepsilon\}$, where $\mathscr{O} = Y \cup \{\delta\}$ is the observation alphabet, that projects recognizer transitions onto observable symbols (output changes and time-tick events). The formal definition of this mapping is given in Definition 5.14.

The concurrent composition ensures that the fault monitor tracks fault occurrences in lockstep with the EAF: any transition labeled f causes the monitor component to enter and remain in state F, thereby annotating all subsequent states as faulty. Importantly, the composition preserves the behavioral language: $L(\text{Rec}(G_{ef})) = L(G_{ef})$, since the monitor does not restrict which transitions can occur—it only augments states with fault labels. The state space grows linearly: $|X_R| = 2|Q_e|$, as each EAF state is duplicated with labels N and F.

Example 5.6 (Fault Recognizer Construction)**.** *Consider the EAF from Example 5.5 with 17 logical states. The fault recognizer $Rec(G_{ef})$ has state set $X_R = Q_e \times \{N,F\}$, yielding $|X_R| = 2 \times 17 = 34$ states. We illustrate the construction with representative states and transitions:*

State construction. *Each EAF state is paired with both fault labels:*

- *EAF state $(x_0,1)_0$ produces recognizer states $((x_0,1)_0,N)$ and $((x_0,1)_0,F)$*
- *EAF state $(x_1,1)_1$ produces recognizer states $((x_1,1)_1,N)$ and $((x_1,1)_1,F)$*
- *Similarly for all 17 EAF states*

The initial recognizer state is $x_{R,0} = ((x_0,1)_0,N)$, pairing the initial EAF state with the normal health status.

Transition construction. *Each EAF transition is duplicated for both monitor states, with the fault label updated according to δ_M:*

Time-driven transitions (label n):

- *EAF transition $(x_0,1)_0 \xrightarrow{n} (x_0,1)_1$ produces:*
  - $((x_0,1)_0,N) \xrightarrow{n} ((x_0,1)_1,N)$ *(normal stays normal: $\delta_M(N,n) = N$)*
  - $((x_0,1)_0,F) \xrightarrow{n} ((x_0,1)_1,F)$ *(faulty stays faulty: $\delta_M(F,n) = F$)*

Nominal transitions (label n):

- *EAF transition $(x_0,1)_1 \xrightarrow{n} (x_1,1)_0$ produces:*
  - $((x_0,1)_1,N) \xrightarrow{n} ((x_1,1)_0,N)$ *(normal stays normal)*
  - $((x_0,1)_1,F) \xrightarrow{n} ((x_1,1)_0,F)$ *(faulty stays faulty)*

Fault transitions (label f):

- *EAF transition $(x_1,1)_1 \xrightarrow{f} (x_0,1)_0$ (fault at $j = 1$) produces:*
  - $((x_1,1)_1,N) \xrightarrow{f} ((x_0,1)_0,F)$ *(normal becomes faulty: $\delta_M(N,f) = F$)*
  - $((x_1,1)_1,F) \xrightarrow{f} ((x_0,1)_0,F)$ *(faulty stays faulty: $\delta_M(F,f) = F$)*

Fault tracking. *The monitor component ensures that once the system transitions from normal to faulty status (via any f-labeled transition from a state with monitor label N to one with label F), all subsequent states in that path carry the label F. For instance, the run:*

$$((x_0,1)_0,N) \xrightarrow{n} ((x_0,1)_1,N) \xrightarrow{n} ((x_1,1)_0,N) \xrightarrow{n} ((x_1,1)_1,N) \xrightarrow{f} ((x_0,1)_0,F) \xrightarrow{n} ((x_0,1)_1,F)$$

demonstrates that after the fault event at step 5, the system remains in the faulty region (label F) despite subsequent nominal transitions.

The total transition count is $|\Delta_R| = 2|\Delta|$, where each of the EAF's time, nominal, and fault transitions is doubled. This explicit fault tracking enables the subsequent diagnoser construction to distinguish normal from faulty behaviors based solely on observation sequences. The complete fault recognizer structure with all 34 states is shown in Figure 5.7.

5.3.3.3 Observation Mappings

The fault recognizer tracks all possible system evolutions with explicit fault status. However, an external observer can only perceive outputs and timing information, not the internal state transitions or fault occurrences. To bridge the SOAF's continuous-time behaviors and the recognizer's discrete-time transitions, we introduce two complementary observation mappings: (i) the *logical observation mapping* ψ that discretizes continuous SOAF behaviors into observation sequences, and (ii) the *observation mapping* $\mathscr{P}$ that projects recognizer transitions onto the observable alphabet. These mappings allow us to formally relate physical observations from the SOAF to logical observations from the recognizer.

Logical Observation Mapping. To perform fault diagnosis on the SOAF, we must abstract the continuous output behaviors into discrete logical observations:

Definition 5.13 (Logical Observation Mapping)**.** *Given a SOAF $G_f = \langle G, B_f, Q_f, \mathscr{I} \rangle$ with output behaviors $L_\delta(G_f)$, the logical observation mapping $\psi : L_\delta(G_f) \to (Y \cup \{\delta\})^*$ maps each output behavior to an observation sequence. For an output behavior $\omega = (y_0,t_0)(y_1,t_1)\cdots(y_n,t_n) \in L_\delta(G_f)$:*

$$\psi(\omega) = \delta^{k_0}\, y_1\, \delta^{k_1}\, y_2\, \delta^{k_2} \cdots y_n\, \delta^{k_n}$$

where $k_i = \lfloor t_i/\delta \rfloor$ for $i = 0,1,\ldots,n$, and $\lfloor\cdot\rfloor$ denotes the floor function.

This mapping discretizes continuous dwell times into multiples of δ, where each δ symbol represents the passage of minimal dwell time, and output changes are explicitly recorded.

Observation Mapping on Recognizer. We now define how transitions in the fault recognizer map to observable symbols:

Definition 5.14 (Observation Mapping)**.** *Given a fault recognizer $Rec(G_{ef}) = (X_R, \Sigma, \Delta_R, x_{R,0})$, the observation mapping is a homomorphism $\mathscr{P} : \Delta_R \to \mathscr{O} \cup \{\varepsilon\}$ where $\mathscr{O} = Y \cup \{\delta\}$ is the*

observation alphabet. For each transition $\tau = ((q,\gamma),a,(q',\gamma')) \in \Delta_R$ *with* $q = (x_1,y_1)_j$ *and* $q' = (x_2,y_2)_{j'}$*:*

$$\mathscr{P}(\tau) = \begin{cases} y_2 & \text{if } y_1 \neq y_2 \text{ (output change)} \\ \delta & \text{if } y_1 = y_2 \text{ and } (x_1,y_1) = (x_2,y_2) \text{ and } (j' = j+1 \text{ or } j' = j) \text{ (time progression)} \\ \varepsilon & \text{if } y_1 = y_2 \text{ and } (x_1,y_1) \neq (x_2,y_2) \text{ (silent transition)} \end{cases} \tag{5.7}$$

The mapping extends to sequences $\mathscr{P} : \Delta_R^* \to (\mathscr{O} \cup \{\varepsilon\})^*$ as a homomorphism:

- $\mathscr{P}(\varepsilon) = \varepsilon$ where ε is the empty sequence
- $\mathscr{P}(\tau \cdot s) = \mathscr{P}(\tau) \cdot \mathscr{P}(s)$ for $\tau \in \Delta_R$ and $s \in \Delta_R^*$

This homomorphism property ensures that the observation of a transition sequence equals the concatenation of individual transition observations, preserving the compositional structure of system behaviors.

To obtain observable sequences without silent transitions, we define the ε-erasure operation that removes all ε symbols from a sequence:

Definition 5.15 (ε-Erasure)**.** *For a sequence* $w \in (\mathscr{O} \cup \{\varepsilon\})^*$*, the* ε*-erasure* $erase_\varepsilon(w) \in \mathscr{O}^*$ *is defined recursively as:*

$$\begin{aligned} erase_\varepsilon(\varepsilon) &= \varepsilon \quad \textit{(empty sequence)} \\ erase_\varepsilon(o \cdot w) &= o \cdot erase_\varepsilon(w) \quad \textit{for } o \in \mathscr{O} \\ erase_\varepsilon(\varepsilon \cdot w) &= erase_\varepsilon(w) \quad \textit{(remove unobservable symbol)} \end{aligned}$$

This operation removes all ε *symbols while preserving the order of observable symbols.*

Consistency between Mappings. The following proposition establishes that the two observation mappings ψ and $\mathscr{P}$ produce consistent results: physical observations from the SOAF match logical observations from the recognizer when related through δ-quantization and ε-erasure.

Proposition 5.2 (Observation Consistency)**.** *Let* G_f *be a SOAF,* G_{ef} *its corresponding EAF, and* $Rec(G_{ef})$ *the fault recognizer. For any output behavior* $\omega \in L_\delta(G_f)$*, there exists a run* ρ *of the recognizer such that:*

$$erase_\varepsilon(\mathscr{P}(\rho)) = \psi(Q_\delta(\omega))$$

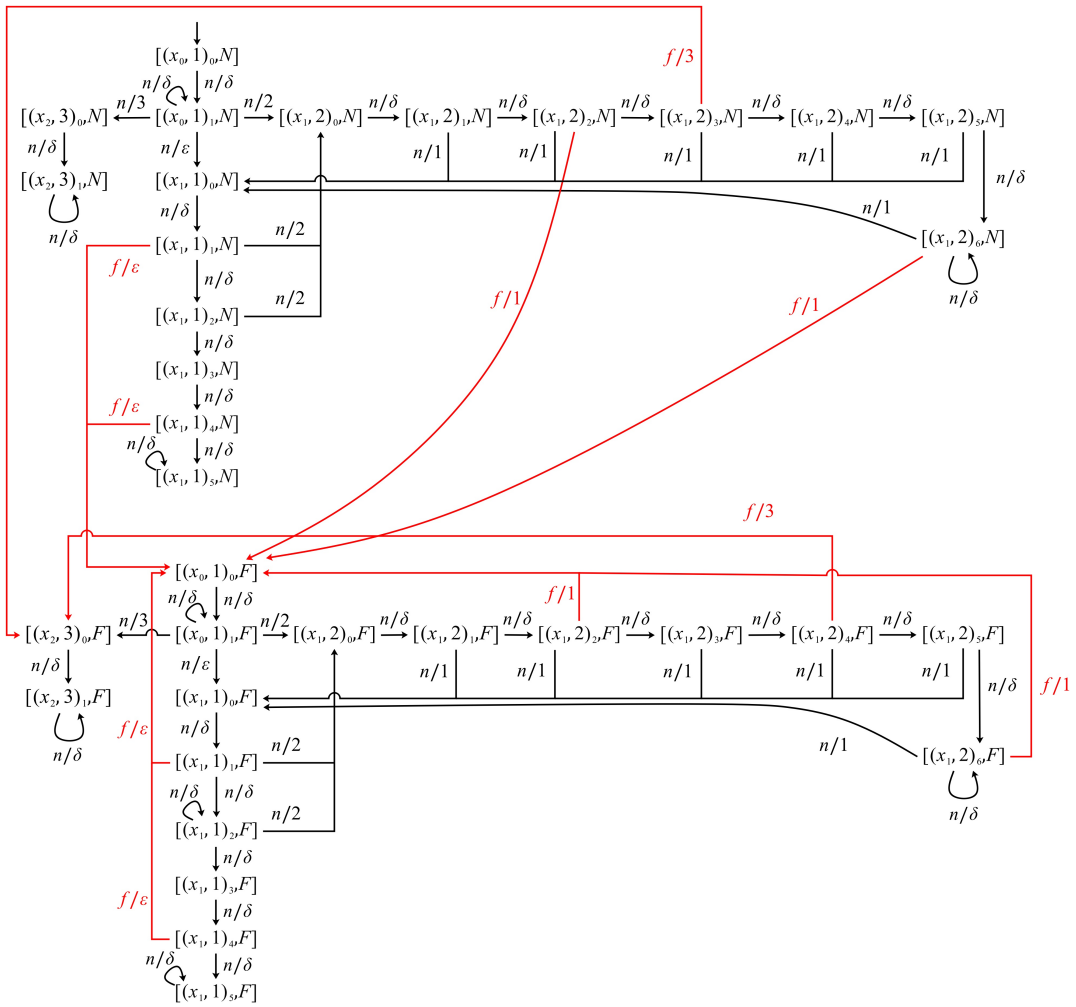


Figure 5.7: Complete fault recognizer with observation mappings.

where ψ is the logical observation mapping, Q_δ is the δ-quantization (Definition 5.10), $\mathscr{P}$ is the observation mapping on the recognizer, and erase$_\varepsilon$ is the ε-erasure operation (Definition 5.15). Equivalently, after removing silent transitions, the recognizer produces observations consistent with the δ-quantized behaviors of the SOAF.

Proof. By Theorem 5.1, for each $\omega \in L_\delta(G_f)$, there exists a corresponding run ρ_{ef} in the EAF that generates $Q_\delta(\omega)$. Since $L(\mathrm{Rec}(G_{ef})) = L(G_{ef})$, this run extends to a run ρ in the recognizer by adding fault monitor state components, preserving the transition structure.

We verify that $\mathrm{erase}_\varepsilon(\mathscr{P}(\rho)) = \psi(Q_\delta(\omega))$ by analyzing how each segment of ω maps to observations:

Time-driven transitions. For each dwell period of duration τ_i in global state (x,y), let $k_i = \lfloor \tau_i/\delta \rfloor \geq 1$. The δ-quantized behavior contains the pair $(y, k_i\delta)$. The corresponding EAF run includes k_i consecutive time-driven transitions. If $k_i \leq \mathscr{R}(x,y)$, these are:

$$(x,y)_0 \xrightarrow{n} (x,y)_1 \xrightarrow{n} \cdots \xrightarrow{n} (x,y)_{k_i-1} \xrightarrow{n} (x,y)_{k_i}$$

If $k_i > \mathscr{R}(x,y)$, the run reaches $(x,y)_{\mathscr{R}(x,y)}$ and accrues the remaining $k_i - \mathscr{R}(x,y)$ steps via the self-loop:

$$(x,y)_0 \xrightarrow{n} \cdots \xrightarrow{n} (x,y)_{\mathscr{R}(x,y)} \xrightarrow{n} (x,y)_{\mathscr{R}(x,y)} \xrightarrow{n} \cdots \xrightarrow{n} (x,y)_{\mathscr{R}(x,y)}$$

By Definition 5.14, each transition (whether $j' = j+1$ or $j' = j$ for self-loops) satisfies $(x_1,y_1) = (x_2,y_2)$, yielding $\mathscr{P}(\tau) = \delta$. Thus, these k_i transitions each produce observation δ, contributing the segment δ^{k_i} to the overall observation sequence.

Nominal transitions. Each nominal arc traversal from (x,y) to $(\bar{x},\bar{y})$ with $\bar{y} \neq y$ corresponds to a transition $(x,y)_j \xrightarrow{n} (\bar{x},\bar{y})_0$ in the EAF, where $j = \min\{k_i, \mathscr{R}(x,y)\}$ is the index reached after the preceding dwell period. Since $y_1 = y \neq \bar{y} = y_2$, the observation mapping yields $\mathscr{P}(\tau) = \bar{y}$, consistent with the output change recorded in ω.

Fault transitions. Each fault arc traversal from (x,y) to (x',y') produces a transition $(x,y)_j \xrightarrow{f} (x',y')_0$, where j is the appropriate time index at which the fault occurs (determined by the fault time mapping $\mathscr{I}$). If $y' \neq y$, then $\mathscr{P}(\tau) = y'$; if $y' = y$, then $\mathscr{P}(\tau) = \varepsilon$ (silent).

Sequence composition. Let $\omega = (y_0,\tau_0)(y_1,\tau_1)\cdots(y_n,\tau_n)$ be a SOAF behavior. The δ-quantization yields $Q_\delta(\omega) = (y_0,k_0\delta)(y_1,k_1\delta)\cdots(y_n,k_n\delta)$ where $k_i = \lfloor \tau_i/\delta \rfloor$ for each i. The logical observation mapping applied to the quantized behavior produces:

$$\psi(Q_\delta(\omega)) = \delta^{k_0} y_1 \delta^{k_1} y_2 \delta^{k_2} \cdots y_n \delta^{k_n}$$

The recognizer run ρ generates exactly this sequence when applying $\mathscr{P}$ and removing all ε symbols, since:

- Time-driven transitions (including self-loops) contribute the δ^{k_i} segments
- Output-changing transitions contribute the y_i symbols
- Silent transitions (if any) produce ε symbols that are removed by erase_ε

This establishes $\text{erase}_\varepsilon(\mathscr{P}(\rho)) = \psi(Q_\delta(\omega))$, confirming the consistency between the δ-quantized physical observations in the SOAF and logical observations in the recognizer after ε-erasure. □

Example 5.7 (Observation Consistency Verification)**.** *Consider the SOAF from Example 5.5 with minimal dwell time $\delta = 1$. Suppose the SOAF produces the following output behavior:*

$$\omega = (1, 1.5)(3, 5.4)$$

indicating output 1 for 1.5 seconds, then output 3 for 5.4 seconds.

Right-hand side: $\psi(Q_\delta(\omega))$

First, apply δ-quantization with $\delta = 1$:

$$Q_\delta(\omega) = (1, 1 \cdot \delta)(3, 5 \cdot \delta) = (1,1)(3,5)$$

where $k_0 = \lfloor 1.5/1 \rfloor = 1$ and $k_1 = \lfloor 5.4/1 \rfloor = 5$.

Then apply the logical observation mapping ψ:

$$\psi(Q_\delta(\omega)) = \delta^1 \cdot 3 \cdot \delta^5 = \delta\, 3\, \delta^5$$

Left-hand side: $\textbf{\textit{erase}}_\varepsilon(\mathscr{P}(\rho))$

The corresponding recognizer run starts from $((x_0, 1)_0, N)$ and includes:

$((x_0,1)_0,N) \xrightarrow{n} ((x_0,1)_1,N)$	*time progression:* $\mathscr{P} = \delta$
$((x_0,1)_1,N) \xrightarrow{n} ((x_2,3)_0,N)$	*output change to 3:* $\mathscr{P} = 3$
$((x_2,3)_0,N) \xrightarrow{n} ((x_2,3)_1,N)$	*time progression:* $\mathscr{P} = \delta$
$((x_2,3)_1,N) \xrightarrow{n} ((x_2,3)_1,N)$	*self-loop:* $\mathscr{P} = \delta$
(three more self-loops)	$\mathscr{P} = \delta\delta\delta$

Concatenating these observations:

$$\mathscr{P}(\rho) = \delta \cdot 3 \cdot \delta^5$$

Since this run contains no silent (ε-labeled) transitions, ε-erasure has no effect:

$$erase_\varepsilon(\mathscr{P}(\rho)) = \mathscr{P}(\rho) = \delta \cdot 3 \cdot \delta^5$$

Verification: *$erase_\varepsilon(\mathscr{P}(\rho)) = \psi(Q_\delta(\omega)) = \delta\, 3\, \delta^5$, confirming observation consistency.*

5.3.3.4 Complexity Analysis

Proposition 5.3 (Fault Recognizer Complexity)**.** *Given an EAF $G_{ef} = (Q_e, \Sigma, \Delta, q_0)$ with $|Q_e| = n$ states and $|\Delta| = m$ transitions, the fault recognizer $Rec(G_{ef}) = (X_R, \Sigma, \Delta_R, x_{R,0})$ has:*

- ***State space:*** *$|X_R| = 2|Q_e| = 2n$ states (each EAF state paired with N or F label)*
- ***Transition set:*** *$|\Delta_R| = 2|\Delta| = 2m$ transitions (each EAF transition duplicated for both fault labels)*
- ***Construction time:*** *$O(n+m)$ to enumerate all states and transitions*

Proof. **State space.** By Definition 5.12, $X_R = Q_e \times X_M = Q_e \times \{N, F\}$, giving exactly $|X_R| = |Q_e| \times |X_M| = n \times 2 = 2n$ states.

Transition set. The transition relation is $\Delta_R = \{((q,\gamma), a, (q', \delta_M(\gamma, a))) \mid (q, a, q') \in \Delta, \gamma \in X_M\}$. For each of the m EAF transitions $(q, a, q') \in \Delta$ and each of the 2 monitor states $\gamma \in \{N, F\}$, we generate exactly one recognizer transition. Thus $|\Delta_R| = |\Delta| \times |X_M| = m \times 2 = 2m$.

Construction time. The concurrent composition algorithm iterates through all n EAF states to form $2n$ product states, then iterates through all m EAF transitions twice (once for each monitor state) to construct $2m$ product transitions, yielding $O(n+m)$ total construction time. □

5.3.4 Diagnoser Construction and Analysis

The fault recognizer constructed in Section 5.3.3 is nondeterministic due to unobservable (ε-labeled) transitions. To enable online fault diagnosis, we must determinize this automaton while preserving its observation-labeled behaviors. This process yields the *diagnoser*, a deterministic automaton that tracks all possible recognizer states consistent with the observed outputs. The diagnoser construction leverages the observation mapping $\mathscr{P}$ (Definition 5.14) and the logical observation mapping ψ (Definition 5.13) established in Section 5.3.3.

5.3.4.1 ε-Closure

To handle nondeterminism from unobservable transitions, we compute the ε-closure, which captures all states reachable through sequences of silent transitions.

Definition 5.16 (ε-Closure)**.** *Given a fault recognizer $Rec(G_{ef}) = (X_R, \Sigma, \Delta_R, x_{R,0})$ with observation mapping $\mathscr{P}$, the ε-closure of a state $x \in X_R$ is:*

$$\begin{aligned}\varepsilon\text{-}closure(x) = \{x' \in X_R \mid &\exists n \geq 0,\ \exists \tau_1, \ldots, \tau_n \in \Delta_R,\ \exists x_0, \ldots, x_n \in X_R : \\ &x_0 = x,\ x_n = x',\ \tau_i = (x_{i-1}, a_i, x_i) \text{ for } i = 1, \ldots, n, \\ &\text{and } \mathscr{P}(\tau_i) = \varepsilon \text{ for all } i \in \{1, \ldots, n\}\}\end{aligned}$$

where $n = 0$ corresponds to the empty path with $x' = x$, ensuring reflexivity.

For a set of states $S \subseteq X_R$:

$$\varepsilon\text{-}closure(S) = \bigcup_{x \in S} \varepsilon\text{-}closure(x)$$

When the recognizer is in state x, an external observer cannot distinguish x from any state in ε-closure(x), since the transitions connecting them produce no observable output. This set represents the uncertainty arising from partial observability.

Example 5.8 (ε-Closure Computation)**.** *Consider a recognizer with the following states and transitions:*

- $s_1 \xrightarrow{\varepsilon} s_2$ *(silent transition: internal state change with no output change)*
- $s_2 \xrightarrow{\varepsilon} s_3$ *(silent transition: fault occurs but output unchanged)*
- $s_1 \xrightarrow{\delta} s_4$ *(observable transition: time progression)*

The ε-closures are:

- $\varepsilon\text{-}closure(s_1) = \{s_1, s_2, s_3\}$ *(from s_1, the system may silently reach s_2 and then s_3)*
- $\varepsilon\text{-}closure(s_2) = \{s_2, s_3\}$ *(from s_2, only s_3 is reachable via ε)*
- $\varepsilon\text{-}closure(s_3) = \{s_3\}$ *(no outgoing ε-transitions)*
- $\varepsilon\text{-}closure(s_4) = \{s_4\}$ *(no outgoing ε-transitions)*

If an observer knows the system is in state s_1 but has seen no output changes, the system could actually be in any of $\{s_1, s_2, s_3\}$ due to unobservable transitions.

5.3.4.2 Diagnoser Construction

The fault recognizer is nondeterministic due to ε-transitions, making it unsuitable for online diagnosis where a single current state estimate is required. To obtain a deterministic automaton, we apply the classical subset construction (powerset construction) adapted to observation-driven transitions. The key idea is that each diagnoser state represents a *belief state*—a set of recognizer states that are consistent with the observed output sequence. When a new observation arrives, the diagnoser deterministically transitions to a new belief state by: (1) identifying all recognizer states reachable via transitions labeled with that observation, and (2) computing the ε-closure to account for unobservable moves.

Definition 5.17 (Diagnoser)**.** *Given a fault recognizer $Rec(G_{ef})$ with observation mapping $\mathscr{P}$, the diagnoser is the deterministic finite automaton:*

$$G_{diag} = (Z, \mathscr{O}, \delta_{diag}, z_0)$$

where:

- $Z \subseteq 2^{X_R}$ *is the state space, with each state $z \subseteq X_R$ representing a set of possible recognizer states*
- $\mathscr{O} = Y \cup \{\delta\}$ *is the observation alphabet*
- $z_0 = \varepsilon\text{-closure}(\{x_{R,0}\})$ *is the initial state*
- $\delta_{diag} : Z \times \mathscr{O} \to Z$ *is the transition function defined as:*

$$\delta_{diag}(z,o) = \varepsilon\text{-closure}\left(\bigcup_{x \in z} \{x' \mid \exists \tau = (x,a,x') \in \Delta_R \text{ with } \mathscr{P}(\tau) = o\}\right)$$

Each diagnoser state $z = \{(q_1, \gamma_1), \ldots, (q_k, \gamma_k)\}$ maintains the set of all EAF states q_i and fault labels $\gamma_i \in \{N, F\}$ consistent with the observation history, ensuring that the diagnoser tracks all possible recognizer configurations at each step.

Construction Algorithm. The diagnoser is constructed incrementally via breadth-first exploration of reachable belief states:

The algorithm constructs the diagnoser incrementally via breadth-first exploration, terminating when all reachable belief states have been processed.

Algorithm 7 Diagnoser Construction via Subset Construction

Require: Fault recognizer $\mathrm{Rec}(G_{ef}) = (X_R, \Sigma, \Delta_R, x_{R,0})$, observation mapping $\mathscr{P}$
Ensure: Diagnoser $G_{diag} = (Z, \mathscr{O}, \delta_{diag}, z_0)$

$z_0 \leftarrow \varepsilon\text{-closure}(\{x_{R,0}\})$ ▷ Initial diagnoser state
$Z \leftarrow \{z_0\}$ ▷ Diagnoser state set
$\delta_{diag} \leftarrow \emptyset$ ▷ Transition function
Queue $\leftarrow \{z_0\}$ ▷ States to process
while Queue $\neq \emptyset$ **do**
    $z \leftarrow$ Queue.dequeue()
    for each $o \in \mathscr{O}$ **do** ▷ For each observable
        Next $\leftarrow \emptyset$
        for each $x_r \in z$ **do** ▷ $x_r = (q, \gamma) \in X_R$
            for each transition $(x_r, a, x_r') \in \Delta_R$ with $\mathscr{P}((x_r, a, x_r')) = o$ **do**
                Next $\leftarrow$ Next $\cup \{x_r'\}$ ▷ x_r' already contains fault label
            end for
        end for
        if Next $\neq \emptyset$ **then**
            $z' \leftarrow \varepsilon\text{-closure}(\text{Next})$
            $\delta_{diag}(z, o) \leftarrow z'$
            if $z' \notin Z$ **then**
                $Z \leftarrow Z \cup \{z'\}$
                Queue.enqueue(z')
            end if
        end if
    end for
end while
return $(Z, \mathscr{O}, \delta_{diag}, z_0)$

Complexity. The diagnoser construction has worst-case exponential complexity in the number of recognizer states. Specifically, given a recognizer with $|X_R| = n$ states and $|\Delta_R| = m$ transitions:

- **State space:** $|Z| \leq 2^n$ in the worst case, as each diagnoser state is a subset of X_R
- **Transition computation:** For each state $z \in Z$ and each observation $o \in \mathscr{O}$, computing $\delta_{diag}(z,o)$ requires examining up to $|z| \times m$ transitions and computing ε-closure, yielding $O(|z| \times m + n^2)$ per transition
- **Total time:** $O(2^n \times |\mathscr{O}| \times (n \times m + n^2)) = O(2^n \times n^2 \times m)$ in the worst case

In practice, the actual number of reachable belief states is often much smaller than 2^n, especially when the system has strong observability properties. The exponential growth is unavoidable for general nondeterministic automata, but techniques such as state merging and on-the-fly construction can mitigate the state explosion in many practical cases.

5.3.4.3 Diagnostic Evaluation Function

The diagnoser enables real-time fault detection by evaluating the fault status of reachable state sets:

Definition 5.18 (Diagnostic Evaluation)**.** *For any non-empty diagnoser state $z \in Z$ where $z = \{(q_1, \gamma_1), \ldots, (q_k, \gamma_k)\}$, the diagnostic evaluation function $\varphi : Z \setminus \{\emptyset\} \to \{N, F, U\}$ is defined as:*

$$\varphi(z) = \begin{cases} N & \text{if } \forall (q_i, \gamma_i) \in z : \gamma_i = N \text{ (certainly normal)} \\ F & \text{if } \forall (q_i, \gamma_i) \in z : \gamma_i = F \text{ (certainly faulty)} \\ U & \text{otherwise (uncertain)} \end{cases}$$

This three-valued logic explicitly captures diagnostic certainty, enabling the diagnoser to distinguish between confirmed diagnoses (N or F) and ambiguous observations (U) that require additional observations for resolution.

Proposition 5.4 (Diagnoser Correctness)**.** *Given a SOAF $G_f = \langle G, B_f, Q_f, \mathscr{I} \rangle$ and its diagnoser G_{diag}, let $u \in \mathscr{O}^*$ be a logical observation sequence and $z = \delta_{diag}(z_0, u)$ be the diagnoser state reached after observing u. The diagnostic evaluation $\varphi(z)$ is both sound and complete with respect to the actual system behaviors:*

1. ***(Soundness)** If $\varphi(z) = N$, then no SOAF behavior producing observation u contains a fault. If $\varphi(z) = F$, then every SOAF behavior producing observation u contains a fault.*

2. ***(Completeness)*** *The diagnoser state z contains all and only those recognizer states reachable through behaviors producing observation u, ensuring no actual behaviors are missed and no spurious behaviors are introduced.*

3. ***(Monotonicity)*** *Once $\varphi(z) = F$ is reached, all subsequent states z' reachable from z satisfy $\varphi(z') = F$ (faults are permanent).*

Proof. By the diagnoser construction (Definition 5.17), $z = \delta_{diag}(z_0, u)$ contains all recognizer states reachable through runs producing observation u. Since $L(\mathrm{Rec}(G_{ef})) = L(G_{ef})$ and by Theorem 5.1, each recognizer run corresponds to a SOAF behavior ω with $\psi(Q_\delta(\omega)) = u$. We establish each property using this correspondence.

(1) Soundness. By Definition 5.18, $\varphi(z) = N$ implies all $(q_i, \gamma_i) \in z$ have $\gamma_i = N$, while $\varphi(z) = F$ implies all have $\gamma_i = F$. By the fault monitor construction (Definition 5.11), label N indicates no fault transition has occurred, while label F (which is absorbing) indicates a fault transition has occurred. Via the recognizer-SOAF correspondence, this translates directly to the presence or absence of fault arcs in the corresponding SOAF behaviors.

(2) Completeness. The subset construction with ε-closure ensures z contains all recognizer states reachable through runs producing u (no behaviors missed) and only such states (no spurious behaviors). By the recognizer-SOAF correspondence established above, z captures exactly those SOAF behaviors consistent with observation u.

(3) Monotonicity. Once $\varphi(z) = F$, all states in z have fault label F. By the fault monitor's absorbing property (Definition 5.11), any successor state $z' = \delta_{diag}(z, o)$ will contain only states with label F, yielding $\varphi(z') = F$. □

Example 5.9 (Fault Diagnosis Application)**.** *Consider the SOAF from Example 5.5 and its corresponding diagnoser. A partial view of the diagnoser structure is shown in Figure 5.8. We analyze three scenarios to demonstrate the diagnostic capability: certain normal, certain faulty, and uncertain diagnoses.*

Scenario 1: Certain normal diagnosis. *Let $\omega_1 = (1, 1.5)(3, 5.4)$ be an output behavior. The corresponding logical observation sequence is $u_1 = \psi(\omega_1) = \delta\, 3\, \delta^5$. The diagnoser processes this sequence as follows:*

- *Initial: $z_0 = \varepsilon\text{-closure}(\{((x_0, 1)_0, N)\})$*
- *After δ: $z_1' = \delta_{diag}(z_0, \delta) = \{((x_0, 1)_1, N), (x_1, 1)_0, N)\}$ (time progression in initial state)*
- *After 3: $z_1'' = \delta_{diag}(z_1', 3) = ((x_2, 3)_0, N)$ (output change to 3, transition to state with output 3)*

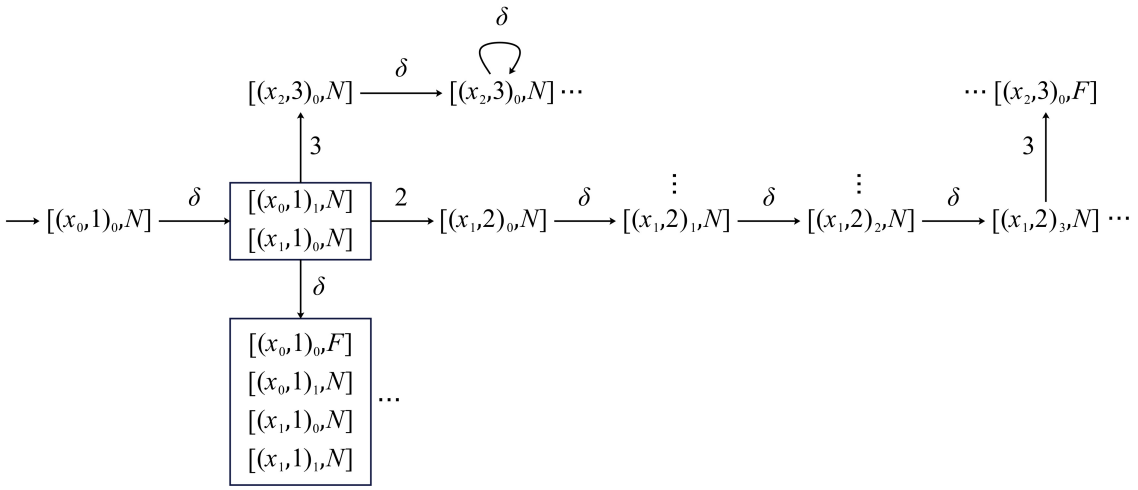


Figure 5.8: Diagnoser for the SOAF from Example 5.5 (partial view showing representative belief states and diagnostic labels).

- *After* δ^5*:* $z_1 = \delta_{diag}(z_1'', \delta^5) = \{((x_2,3)_1,N)\}$ *(time progression, remaining in normal region)*

Since z_1 *contains only states with normal label N, we obtain* $\varphi(z_1) = N$*:* no fault has occurred.

Scenario 2: Certain faulty diagnosis. *Let* $\omega_2 = (1,1.9)(2,3.3)(3,0.8)$ *with observation sequence* $u_2 = \psi(\omega_2) = \delta\,2\,\delta^3\,3$*. Processing this sequence:*

- *After* $\delta\,2$*: the system transitions to a state with output 2, which corresponds to a fault-prone state*
- *After* δ^3*: dwell time in fault-prone state triggers fault window*
- *After final* 3*:* $z_2 = \delta_{diag}(z_0, u_2) = \{((x_2,3)_0,F)\}$ *(all reachable states have fault label F)*

Since z_2 *contains only states with fault label F, we obtain* $\varphi(z_2) = F$*:* a fault has occurred with certainty.

Scenario 3: Uncertain diagnosis. *Consider a partial observation* $u_3 = \delta^2$ *in a system where multiple paths (both normal and fault-prone) can produce the same initial observations. In such cases, the diagnoser state may contain a mix of states:*

$$z_3 = \delta_{diag}(z_0, \delta) = \{((x_0,1)_1,N),((x_0,1)_0,F),((x_1,1)_0,N),((x_1,1)_1,N)\}$$

where some states have label N and others have label F, indicating that different behaviors (normal and faulty) remain consistent with the observation. This yields $\varphi(z_3) = U$*, indicating* diagnostic uncertainty *that requires additional observations to resolve.*

Summary: *This example demonstrates the three-valued diagnostic logic:* $\varphi(z_1) = N$ *(certain normal),* $\varphi(z_2) = F$ *(certain faulty), and* $\varphi(z_3) = U$ *(uncertain). The diagnoser provides deterministic state updates while explicitly representing diagnostic uncertainty when ambiguity exists.*

5.4 Chapter Summary

This chapter presented a diagnosis approach for dwell-time–triggered faults under output-and-duration observations. We extended SOA to SOAF by separating nominal and fault arcs and attaching time windows that gate fault enablement. The minimal dwell time δ provides a natural time quantum, enabling a finite evolution automaton that preserves window boundaries. From this abstraction, an observer-style pipeline—time discretization, fault-annotated evolution automaton, fault recognizer, and subset determinization—produces a deterministic online diagnoser that classifies runs as normal, faulty, or uncertain with bounded delay. The method restores decidability and follows the familiar complexity profile of observer constructions (linear growth before determinization, possible exponential blow-up after), while aligning with field observability. Overall, the framework serves as an upper-layer safety diagnoser that complements continuous models, providing timing-aware guarantees on finite structures and a clear path to implementation.

Chapter 6

Case Studies

This chapter demonstrates the practical applicability of the Switching Output Automaton (SOA) framework through three comprehensive case studies drawn from safety-critical cyber-physical systems. The first two case studies apply the opacity verification frameworks from Chapter 4: Section 6.1 demonstrates current-state opacity verification for critical infrastructure protection, while Section 6.2 demonstrates timed opacity verification for healthcare privacy. The third case study (Section 6.3) applies the timed fault diagnosis framework from Chapter 5, demonstrating online detection of thermal runaway faults in battery management systems.

All three case studies share a common theme: reasoning about security and reliability properties based solely on observable outputs and their durations, without access to internal discrete states or precise continuous-time measurements. The minimal dwell time δ provides natural discretization that aligns with physical sensor sampling rates and actuator response times, while the output nondeterminism captures measurement aggregation and abstraction typical of real-world monitoring infrastructure.

6.1 Smart Water Supply System: Current State Opacity Verification

Water distribution systems exemplify safety-critical cyber-physical infrastructure where operational transparency conflicts with security requirements. SCADA controllers must continuously monitor water levels and actuator states to enforce safety constraints, yet these observations create potential side channels that can expose operational vulnerabilities. Power-consumption monitoring poses a particularly insidious threat [85], [86]: by passively measuring aggregate electrical load, attackers can infer when critical storage tanks run dangerously low and time coordinated disruptions to maximize impact.

We apply the current-state opacity verification framework from Chapter 4 to the Amira DTS200 testbed [87], a laboratory-scale smart water supply system comprising three storage tanks, two pumps, and three valves. An adversary observes only the aggregate power consumption. We abstract the continuous hybrid dynamics into a 20-state switching output automaton with 9 discrete power levels, where multiple actuator configurations yield identical measurements and thereby induce observational ambiguity.

The verification confirms that the testbed is current-state opaque with respect to critical states: adversaries monitoring power consumption cannot conclusively infer whether the customer supply tank is below the safety threshold. This demonstrates that aggregate power measurements preserve confidentiality of security-sensitive operational states despite continuous monitoring.

6.1.1 System Description and Critical States

Hydraulic architecture and instrumentation

The hydraulic topology comprises three water storage tanks (upstream tanks T_1 and T_3, customer supply tank T_2), two pumps (P_1 and P_2), and three valves (V_1, V_2, V_3), as illustrated in Fig. 6.1. The platform integrates a SCADA supervisory control system that monitors water levels, executes control logic on a programmable logic controller (PLC), and issues actuation commands to pumps and valves. All actuator electrical feeds are routed through a centralized power transducer that measures the aggregate instantaneous power consumption, which constitutes the adversary's sole observation channel.

The key physical and instrumentation components are characterized as follows:

- **Storage tanks.** Each of the three reservoirs exhibits uniform cylindrical geometry with cross-sectional area $A_1 = A_2 = A_3$ and nominal height of 40 cm. Water level in each tank is measured by a calibrated capacitive float sensor, providing continuous level feedback $h_i \in [0,40]$ cm with measurement uncertainty below ± 0.5 cm.

- **Pumping units.** Pumps P_1 and P_2 are binary-actuated centrifugal units with ON/OFF control inputs $q_1, q_2 \in \{0,1\}$. When energized ($q_p = 1$), each pump delivers a nominal volumetric flow rate of $Q_{P_1} = Q_{P_2}$ against the prevailing hydraulic head. Pump dynamics exhibit negligible transient response relative to the 1 Hz control update rate.

- **Control valves.** Valves V_1 (connecting T_1 to T_3) and V_3 (connecting T_3 to T_2) are proportional solenoid valves that accept discrete opening commands $v_1, v_3 \in \{0.5, 1\}$, corresponding to half-open and fully open configurations. Valve V_2, which

couples the customer tank T_2 to the distribution network, is maintained in the fully open configuration $v_2 = 1$ to simulate continuous consumer draw. All valves maintain at least partial opening during normal operation to ensure continuous hydraulic coupling. Flow through each valve is governed by the orifice equation (Torricelli's law) scaled by the commanded opening fraction.

- **Supervisory control logic.** The embedded PLC executes a multi-objective control strategy designed to (i) enforce safety constraints $h_{\min} \leq h_2 \leq h_{\max}$ on the customer tank level; (ii) alternate pump activation ($P_1 \leftrightarrow P_2$) to equalize cumulative operating hours and mitigate mechanical wear; and (iii) modulate valve openings v_1, v_3 to maintain hydraulic balance across the upstream tanks. Each distinct actuator configuration—defined by the tuple (q_1, q_2, v_1, v_3)—induces a characteristic aggregate power consumption level (detailed in the output mapping below).

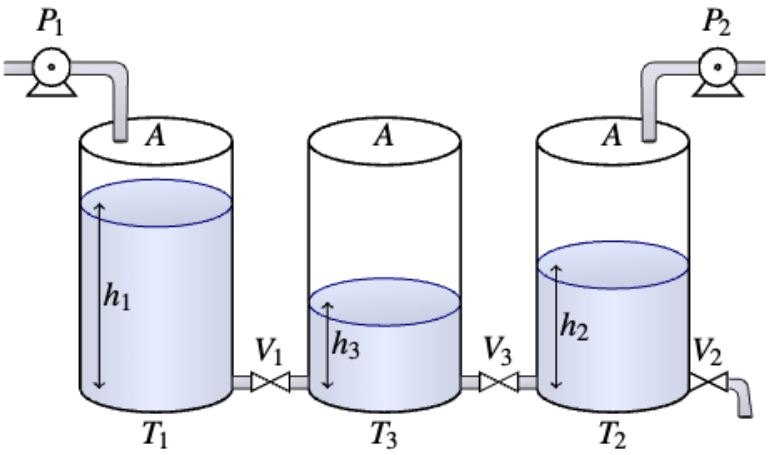


Figure 6.1: Schematic diagram of the Amira DTS200 testbed.

Table 6.1: Physical and operational parameters of the DTS200 experimental platform [87], [88].

Quantity	**Value**
Tank cross-sectional area $A_1 = A_2 = A_3$	154 cm^2
Pump nominal flow $Q_{P_1} = Q_{P_2}$	100 cm^3/s
Valve pipe cross-section $s_{V_1} = s_{V_3}$	5 cm^2
Flow coefficient $a_{V_1} = a_{V_3}$	1
Gravity constant g	981 cm/s^2

Continuous-time hydraulic dynamics

The evolution of water levels in the three-tank system is governed by mass conservation laws. Let $h_i(t)$ denote the water level in tank T_i at time t. Applying the principle of mass balance to each reservoir yields

$$\dot{h}_i = \frac{1}{A_i}\Big(\sum \text{inflows} - \sum \text{outflows}\Big), \quad i \in \{1,2,3\}, \tag{6.1}$$

where A_i is the tank cross-sectional area and the summations account for all volumetric flow rates entering and leaving the respective tank. Substituting the specific hydraulic connectivity of the DTS200 topology, we obtain the following coupled nonlinear ordinary differential equations:

$$\dot{h}_1 = \frac{1}{A_1}\big(q_{P_1} - q_{V_1}\big), \tag{6.2}$$

$$\dot{h}_2 = \frac{1}{A_2}\big(q_{P_2} + q_{V_3} - q_{V_2}\big), \tag{6.3}$$

$$\dot{h}_3 = \frac{1}{A_3}\big(q_{V_1} - q_{V_3}\big). \tag{6.4}$$

Here, q_{P_p} denotes the volumetric flow delivered by pump P_p, and q_{V_k} represents the flow through valve V_k.

Pump flows are modeled as binary switching functions:

$$q_{P_p} = q_p\, Q_{P_p}, \quad q_p \in \{0,1\},$$

where q_p is the discrete ON/OFF command and $Q_{P_p} = 100\ \text{cm}^3/\text{s}$ is the nominal pump capacity.

Valve flows are governed by the orifice equation (Torricelli's law), which relates flow rate to the hydraulic head difference across the valve:

$$q_{V_k} = a_{V_k} s_{V_k}\, v_k\, \operatorname{sgn}(\Delta h_k)\sqrt{2g\,|\Delta h_k|} = c_{V_k}\, v_k\, \operatorname{sgn}(\Delta h_k)\sqrt{|\Delta h_k|},$$

where Δh_k is the instantaneous level difference across valve V_k, v_k encodes the opening fraction with $v_k \in \{0.5,1\}$ for controllable valves V_1 and V_3 and $v_2 = 1$ (always fully open) for the customer outlet valve V_2, a_{V_k} is a dimensionless discharge coefficient, s_{V_k} is the valve orifice cross-sectional area, $g = 981\ \text{cm}/\text{s}^2$ is gravitational acceleration, and $c_{V_k} \triangleq a_{V_k} s_{V_k}\sqrt{2g}$ is the empirically calibrated valve conductance. The sign function $\operatorname{sgn}(\cdot)$ accounts for bidirectional flow in the event of level reversals.

The continuous-time model (6.1)–(6.4), combined with the actuator constitutive relations above, forms a piecewise-smooth hybrid dynamical system. The system exhibits mode-dependent behavior determined by discrete actuator states (pump ON/OFF commands) and continuous state-space regions (defined by relative level orderings among tanks). The discrete abstraction presented in Section 6.1.2 partitions the continuous state space along these mode boundaries to construct the finite-state SOA representation.

Critical states

The customer supply tank T_2 directly determines service availability. Water levels in T_2 falling below the safety threshold $h_{\min} = 18$ cm constitute *critical configurations*. We formalize this condition as:

$$h_2 \leq h_{\min}. \tag{6.5}$$

6.1.2 Switching Output Automaton Construction

We model the DTS200 system as a switching output automaton $G = (X, Y, B, h, x_0, y_0)$ constructed by partitioning the continuous state space along relative tank-level orderings and pump activation patterns. The resulting model has $|X| = 20$ discrete states, $|Y| = 9$ observable power levels, and set-valued output mapping $h : X \to 2^Y$ determined by actuator power profiles. We set the initial state as $x_0 = x_{13}$ (configuration with $h_1 = h_2 > h_3$ and both pumps energized) and initial output $y_0 = y_7$ corresponding to 10 W aggregate power consumption, representing typical balanced operation before any critical conditions occur. We assume minimal dwell time $\delta = 3$ s.

We consider a passive adversary who continuously observes aggregate power consumption but cannot access internal SCADA state variables. The opacity verification objective is to ensure that adversaries cannot conclusively infer when the system occupies critical states where customer supply tank T_2 falls below the safety threshold $h_{\min}$. This abstraction preserves the essential observability structure—critically, multiple states map to identical power levels, creating the ambiguity necessary for current-state opacity.

Discrete state-space abstraction. The continuous hybrid dynamics (6.1)–(6.4) are abstracted into a finite discrete state space X via a two-dimensional partitioning scheme:

1. *Relative tank-level orderings*: The continuous state space $(h_1, h_2, h_3) \in \mathbb{R}^3$ is partitioned into regions defined by the qualitative ordering relations among tank levels (e.g., $h_1 < h_3 < h_2$, $h_3 \geq h_2 \geq h_1$, etc.). These orderings determine the active mode of the piecewise dynamics and the direction of inter-tank flows.

2. *Pump activation patterns*: Each partition is further subdivided according to the discrete pump states $(q_1, q_2) \in \{0,1\}^2$, reflecting the binary ON/OFF commands issued by the supervisory controller.

Applying this abstraction methodology to the DTS200 system yields a discrete state space with $|X| = 20$ states. Each discrete state $x_i \in X$ encodes a specific combination of relative level ordering and pump activation pattern, thereby capturing the essential mode structure of the underlying hybrid system. The discrete states mirror the operational logic of the SCADA controller, which alternates pump duty cycles between P_1 and P_2 to balance cumulative run-time, and modulates valves V_1 and V_3 to equilibrate upstream tank levels.

Notably, states x_{17} through x_{20} correspond to configurations in which the customer tank satisfies the criticality condition $h_2 \leq h_{\min}$ (cf. (6.5)). We define the *secret set* as

$$X_s \triangleq \{x_{17}, x_{18}, x_{19}, x_{20}\} \subseteq X.$$

These four secret states are visually distinguished in the SOA graph depicted in Fig. 6.2. The opacity verification objective is to ensure that adversaries observing power consumption cannot conclusively deduce that the system is currently in any state $x \in X_s$. Table 6.2 summarizes the key dimensions of the resulting SOA: 20 states, 4 secret states, and 9 observable power levels.

Table 6.2: Summary of key automaton components for the SWSS discrete abstraction.

Item	**Value**
SOA states $\lvert X\rvert$	20
Critical discrete states $\lvert X_s\rvert$	4 (states x_{17}–x_{20})
Events $\lvert E\rvert$	plant/logic-induced (omitted here; see Fig. 6.2)
Output alphabet $\lvert Y\rvert$	9 discrete power levels $\{y_1 < \cdots < y_9\}$

Table 6.3 provides a complete enumeration of the 20 discrete states, derived through systematic simulation and analysis of the continuous hybrid model (6.1)–(6.4). The second column employs a graphical encoding scheme to compactly represent the relative vertical ordering of tank water levels: horizontal bars positioned at different heights correspond to the levels of h_1, h_3, and h_2, respectively. The third and fourth columns record the binary states of pumps P_1 and P_2, where $q_i \in \{0,1\}$ indicates OFF or ON. For example, state x_1 encodes the configuration $h_1 < h_3 < h_2$ with both pumps simultaneously energized ($q_1 = q_2 = 1$). The graphical representation of h_2 in the critical states x_{17}–x_{20} is emphasized with a double underline rule to visually distinguish the secret states comprising X_s.

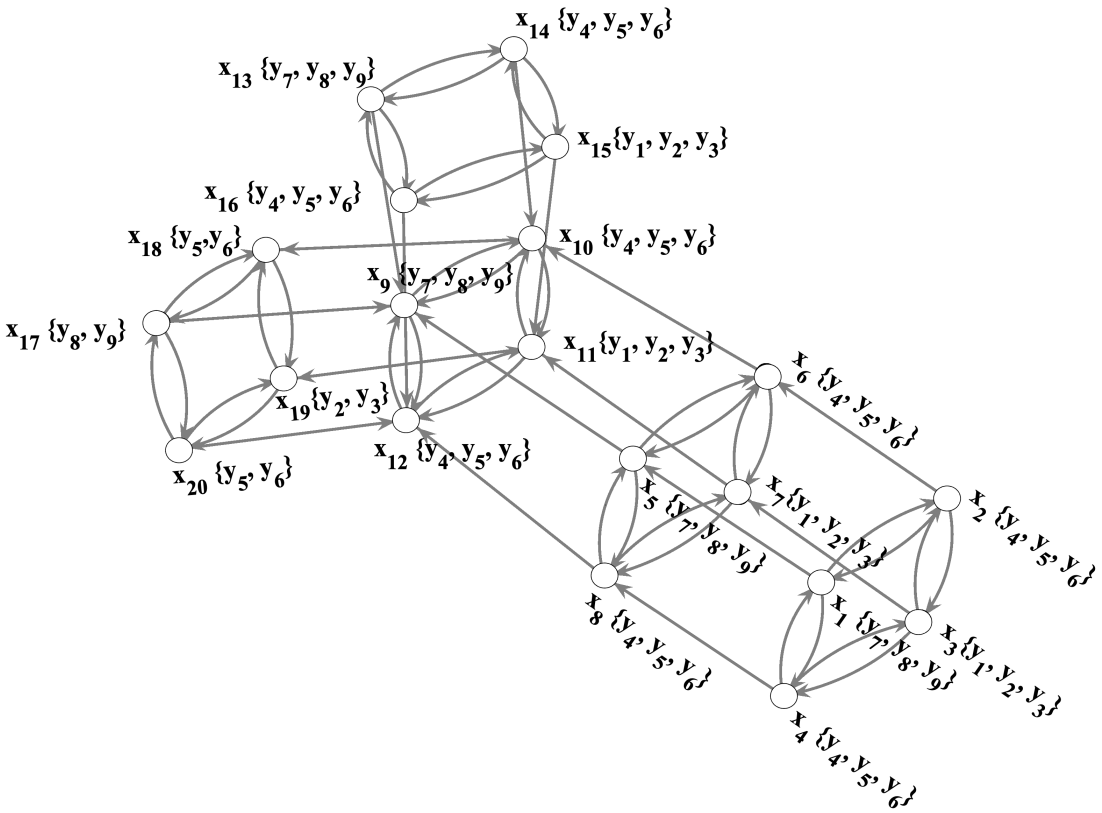


Figure 6.2: Directed graph representation of the 20-state SOA for the DTS200 smart water supply system.

Table 6.3: Discrete states of the SWSS abstraction.

State	h_1 h_3 h_2	q_1	q_2
x_1	_ — ¯	1	1
x_2	_ — ¯	1	0
x_3	_ — ¯	0	0
x_4	_ — ¯	0	1
x_5	_ ¯ _	1	1
x_6	_ ¯ _	1	0
x_7	_ ¯ _	0	0
x_8	_ ¯ _	0	1
x_9	¯ — _	1	1
x_{10}	¯ — _	1	0
x_{11}	¯ — _	0	0
x_{12}	¯ — _	0	1
x_{13}	¯ _ ¯	1	1
x_{14}	¯ _ ¯	1	0
x_{15}	¯ _ ¯	0	0
x_{16}	¯ _ ¯	0	1
x_{17}	¯ — =	1	1
x_{18}	¯ — =	1	0
x_{19}	¯ — =	0	0
x_{20}	¯ — =	0	1

Output mapping and observable power alphabet. The output mapping $h : X \to 2^Y \setminus \{\emptyset\}$ assigns to each discrete state $x \in X$ the set of aggregate power consumption levels that may be observed when the system occupies x. This mapping is determined by the electrical loads of the active actuators: pumps, valves, and ancillary equipment. Empirical measurements on the DTS200 platform yield the following actuator power profiles (Table 6.4):

- Each pump P_i in the ON state draws approximately 3 W;
- Each proportional valve V_1 or V_3 draws 1 W when half-open and 2 W when fully open;
- The customer outlet valve V_2 operates continuously and contributes a constant baseline of 2 W.

Summing the power contributions of all simultaneously active actuators across the various discrete states yields a discrete set of aggregate power levels. Due to the combinatorial structure of actuator configurations, distinct states may share identical total power consumption, inducing a many-to-one output mapping. The aggregate power observations

are quantized into nine discrete levels $y_1 < \cdots < y_9$, which constitute the observable output alphabet Y with $|Y| = 9$. Following the SOA framework, we extend this alphabet with the δ-symbol representing dwell-time elapse, yielding $Y \cup \{\delta\}$ with $|Y \cup \{\delta\}| = 10$ total observable symbols.

Table 6.4: Individual actuator electrical power consumption measured on the DTS200 testbed.

Device	**Electrical power (W)**
Pump P_p (OFF/ON)	$\{0,3\}$
Valve V_i (half-open/fully open)	$\{1,2\}$
Valve V_2 (permanent baseline)	2

Table 6.5 enumerates the nine discrete aggregate power levels that result from combining actuator contributions according to the supervisory control logic. Each row specifies a representative actuator configuration that yields the corresponding power level. Critically, multiple distinct actuator configurations can produce identical aggregate power consumption—for example, $y_5 = 8$ W arises from four different combinations of pump and valve states—thereby creating the observational ambiguity necessary for opacity.

Table 6.5: Discrete aggregate power levels constituting the observable output alphabet Y for the SWSS case study.

Symbol	**Representative actuator configuration**	**Power (W)**
y_1	valves only (both pumps OFF, V_1, V_3 half-open)	4
y_2	valves only (one of V_1, V_3 fully open)	5
y_3	valves only (both V_1, V_3 fully open)	6
y_4	single pump ON, V_1, V_3 half-open	7
y_5	single pump ON, one of V_1, V_3 fully open	8
y_6	single pump ON, both V_1, V_3 fully open	9
y_7	both pumps ON, V_1, V_3 half-open	10
y_8	both pumps ON, one of V_1, V_3 fully open	11
y_9	both pumps ON, V_1, V_3 fully open	12

6.1.3 Verification Procedure and Results

We verify current state opacity following the methodology presented in Chapter 4, Section 4.2. The verification procedure consists of three steps: (1) construct the observer automaton $\mathscr{O}(G)$ via subset construction (Algorithm 3); (2) identify the set Z_s of stable observer states (those for which the δ-transition is not enabled); (3) check whether any stable state $z = (X_z, y) \in Z_s$ satisfies $g(X_z) \subseteq X_s$, where g extracts the set of discrete states

corresponding to the extended states in X_z. The verdict: among 52 reachable observer states, the stable states contain both secret and non-secret discrete states—the DTS200 testbed is **current-state opaque** with respect to critical states $X_s = \{x_{17}, x_{18}, x_{19}, x_{20}\}$.

Verification Results

Applying the current-state opacity verification framework from Chapter 4 to the 20-state SOA yields an evolution automaton G_e with 64 logical states and 324 transitions over the extended output alphabet $Y_e = Y \cup \{\delta\}$. The evolution automaton construction follows Algorithm 4, partitioning each global state (x, y) into waiting and ready logical states to capture the minimal dwell time constraint. Figure 6.3 illustrates part of the evolution automaton structure, showing the initial state (x'_{13}, y_7) and representative transitions.

The observer construction (Algorithm 3) yields an observer automaton $\mathscr{O}(G)$ with 52 reachable states and 199 transitions. The observer tracks sets of consistent logical states through subset construction over the evolution automaton, starting from initial observer state $z_0 = \{(x'_{13}, y_7)\}$. Figure 6.4 presents part of the observer structure, demonstrating how belief states capture observational ambiguity—multiple observer states contain both secret and non-secret discrete states, preventing conclusive inference of critical conditions.

Among the 52 reachable observer states, we identify the stable observer states Z_s by checking which states do not enable δ-transitions. Opacity verification confirms that no stable observer state consists entirely of secret discrete states: for every $z = (X_z, y) \in Z_s$, the set $g(X_z)$ contains at least one non-secret state (i.e., a state in $X \setminus X_s$). Therefore, the DTS200 testbed is **current-state opaque** under passive power observation (Table 6.6).

The key mechanism providing opacity is the output mapping ambiguity: critical states $x_{17}, x_{18}, x_{19}, x_{20}$ produce power consumption levels that are also observable from non-critical states. For example, state x_{17} (critical condition with both pumps ON) may produce the same power level as state x_1 (non-critical condition with both pumps ON and different valve configurations). This observational ambiguity prevents the adversary from conclusively inferring whether the customer supply tank is currently below the safety threshold.

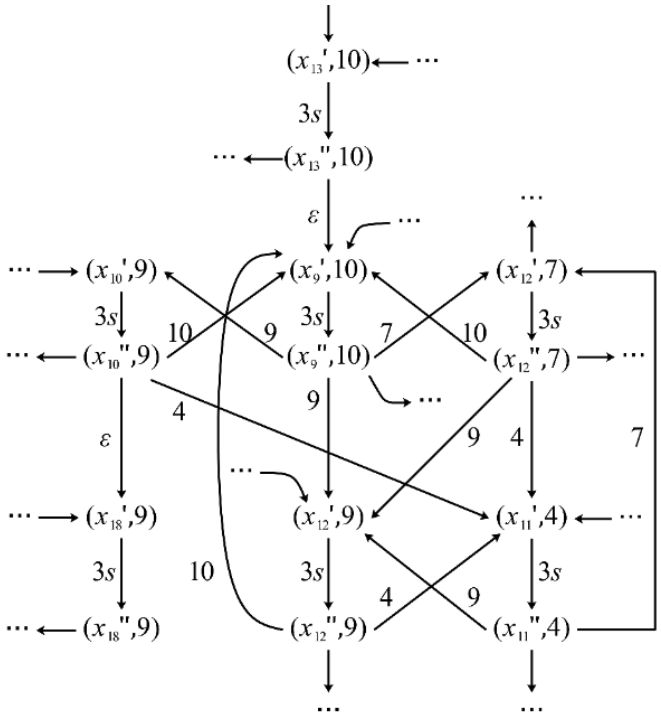


Figure 6.3: Partial structure of the evolution automaton G_e for the DTS200 water supply system.

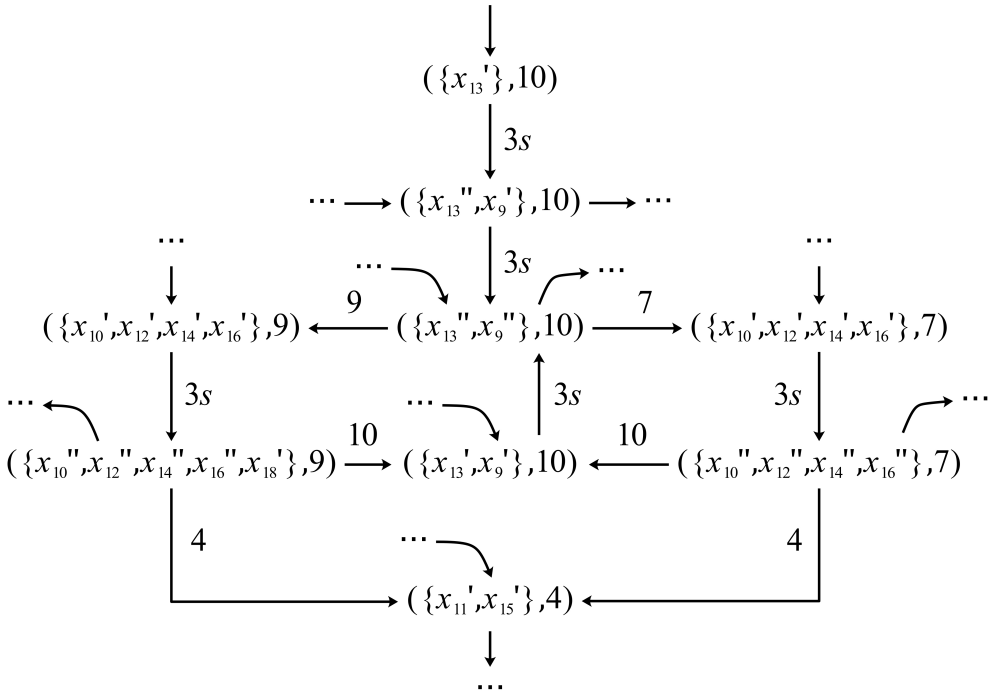


Figure 6.4: Partial structure of the observer automaton $\mathscr{O}(G)$ for the DTS200 water supply system.

Table 6.6: Verification summary: state-space dimensions and opacity verdict for the SWSS case study.

Quantity	Value
SOA discrete states $\lvert X\rvert$	20
Secret states $\lvert X_s\rvert$	4
Output alphabet $\lvert Y\rvert$	9 (power levels)
Extended alphabet $\lvert Y \cup \{\delta\}\rvert$	10
Reachable observer states $\lvert Z\rvert$	52
Observer transitions	199
Current state opacity verdict	OPAQUE (no $z \in Z_s$ with $g(X_z) \subseteq X_s$)

6.2 Patient Monitoring System: Timed Opacity Verification

6.2.1 System Description and Motivation

Modern intensive care units (ICU) increasingly rely on networked patient monitoring systems that continuously track vital signs—heart rate, blood pressure, oxygen saturation—and generate alert signals displayed on nursing station monitors and integrated with electronic health records [89], [90]. While this connectivity improves care coordination and enables data analytics [90], it introduces significant privacy risks [91]–[93]. Hospital networks are vulnerable to data breaches [94], [95], unauthorized access through network monitoring [91], [92], and exploitation of system logs [93], [96] for malicious purposes including insurance fraud [97], targeted attacks on vulnerable patients [98], and unauthorized medical research [99]–[101].

This case study examines the fundamental question: *Can we protect sensitive clinical patterns from unauthorized observers who monitor only the output alert signals and their durations?* Specifically, we consider a scenario where adversaries gain access to the hospital network and observe the timed sequence of alert levels, but cannot directly access internal patient status assessments, state transition events, or operator actions. The challenge is to verify whether the system's inherent observational ambiguity—different internal states producing identical observable outputs—prevents the observer from inferring sensitive temporal patterns that reveal detailed treatment response and prognosis information.

6.2.2 SOA Model

We model the patient monitoring system as an SOA $G = (X, Y, B, h, x_0, y_0)$ that abstracts the complex physiological monitoring and alert generation logic into a discrete-event framework. The system comprises three discrete states representing clinical assessments:

- x_0 (**Stable**): Patient vitals are within normal ranges, requiring routine monitoring;
- x_1 (**Attention**): Minor deviations detected, clinical attention may be needed;
- x_2 (**Critical**): Abnormal vital patterns indicating serious medical concern requiring intervention.

The observable outputs consist of three alert levels displayed on the nursing station monitor and transmitted over the hospital network:

- y_{green}: Green light indicating routine status;
- y_{yellow}: Yellow light indicating caution required;
- y_{red}: Red light indicating immediate intervention needed.

We assume the system starts in the initial state x_0 (Stable) with initial output $y_0 = y_{\text{green}}$, corresponding to the initial global state (x_0, y_{green}), reflecting the typical scenario where a patient enters the monitoring system in stable condition.

The state transition behavior B models clinical progression. From x_0, the patient may transition to x_1 when vital signs show minor deviations, or directly to x_2 in case of sudden deterioration. From x_1, the patient may return to x_0 if vitals improve, or escalate to x_2 if the condition worsens. From x_2, the patient may de-escalate to x_1 after intervention stabilizes the patient, or return to x_0 if the critical event resolves.

The key architectural feature of this monitoring system is the set-valued (nondeterministic) output mapping $h : X \to 2^Y$ that balances clinical accuracy with operational flexibility:

$$\begin{aligned} h(x_0) &= \{y_{\text{green}}\}, \\ h(x_1) &= \{y_{\text{yellow}}\}, \\ h(x_2) &= \{y_{\text{yellow}}, y_{\text{red}}\}. \end{aligned}$$

This mapping encodes the following design principles: when in state x_0 (Stable), only green alerts are displayed; when in state x_1 (Attention), the system displays yellow alerts to ensure appropriate clinical vigilance; when in state x_2 (Critical), the system displays

either yellow (for serious conditions under active management) or red (for emergencies requiring immediate intervention), based on the severity and urgency of the clinical situation as assessed by monitoring algorithms or clinician input.

The resulting global state space is

$$Q = \{(x_0, y_{\text{green}}), (x_1, y_{\text{yellow}}), (x_2, y_{\text{yellow}}), (x_2, y_{\text{red}})\}$$

with cardinality $|Q| = 4$. The complete state transition structure is illustrated in Figure 6.5. Additionally, within state x_2, the system may switch between output values y_{yellow} and y_{red} based on real-time assessment of clinical urgency, vital sign trends, and response to interventions, creating transitions between global states (x_2, y_{yellow}) and (x_2, y_{red}) without changing the discrete state.

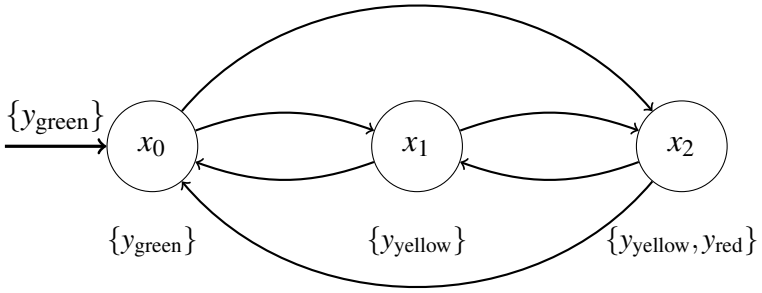


Figure 6.5: SOA model of the patient monitoring system.

The critical observational ambiguity arises from the nondeterministic output mapping: the global state (x_2, y_{yellow}) represents a critical patient displaying a yellow alert, which is observationally indistinguishable from the global state (x_1, y_{yellow}) representing a patient in the attention state. This ambiguity reflects clinical reality: serious conditions under active treatment management (critical state with controlled intervention) may produce the same external alert level as milder abnormalities requiring monitoring (attention state). From a privacy perspective, this inherent observational confusion provides the foundation for protecting sensitive clinical patterns from unauthorized network observers.

Design Rationale and Clinical Validation. The nondeterministic output mapping $h(x_2) = \{y_{\text{yellow}}, y_{\text{red}}\}$ for the critical state reflects the tiered alarm classification systems deployed in real ICU environments [102]. Clinical monitoring systems distinguish between advisory alarms (yellow) generated when vital signs enter physiologically undesirable ranges, and critical alarms (red) indicating potentially dangerous states requiring immediate intervention. An observational analysis of ICU alarm log data reported that 79% of alarms

were yellow and 18% were red [103], supporting the design choice that critical patients under active treatment management may display yellow alerts rather than continuously triggering red alarms.

This observational ambiguity—where yellow alerts may originate from either attention-level conditions (x_1, y_{yellow}) or controlled critical conditions (x_2, y_{yellow})—reflects the clinical reality of alarm fatigue in ICU settings. Research has demonstrated that 72% to 99% of clinical alarms are false [104], contributing to alarm fatigue and its patient safety implications [105]. In the same ICU log analysis, the alarm load averaged 152.5 alarms per bed per day [103]. A systematic review further found that actionable alarms often constitute only a minority of alarms (<1% to 36% across settings) [106]. The prevalence of clinically insignificant alarms means that identical observable signals may correspond to different underlying patient states, providing the foundation for the observational confusion necessary for privacy protection.

The discrete-event system modeling approach is grounded in established practices for healthcare monitoring systems [107], [108]. Patient behavior and clinical progression can be effectively represented as finite state automata, with discrete-event simulation enabling individual-level patient modeling including temporal evolution of clinical courses [108]–[110]. This state-based abstraction integrates naturally with clinical decision support systems that process real-time monitoring data [111], [112].

The focus on protecting temporal patterns—specifically, 20–60 minute durations in the (x_2, y_{yellow}) state—addresses documented privacy concerns regarding treatment response inference [99], [100], [113], [114]. Temporal relationships in health data can reveal sensitive information about treatment efficacy and disease progression [113], [114]. Continuous monitoring via networked sensors raises privacy concerns as temporal data patterns may enable inference of sensitive health information, justifying the need for duration-dependent opacity verification [17], [43], [52], [115].

6.2.3 Timed Secret Specification

We adopt a white-box attacker model following standard opacity verification conventions. The observer (adversary) possesses complete knowledge of the SOA model $G = (X, Y, B, h, x_0, y_0)$, the set of vulnerable states Q_v, and the secret dwell intervals $\mathscr{I}_v$. However, the observer's access is strictly limited to the *output observation channel*, which provides only the timed sequence of observed output values and their durations. The observer cannot directly access the internal state $x(t)$ of the system, state transition events or their timing, or any side channels such as alarm acknowledgment events, operator actions, or system logs. If additional observable events (e.g., alarm confirmations, silence

operations) exist in the actual deployment, they should be incorporated into the output alphabet Y for a more conservative security analysis.

The sensitive information we seek to protect concerns the global state (x_2, y_{yellow}), which represents:

- *Internal assessment*: The system has determined the patient is in a Critical condition (x_2);
- *External display*: A yellow alert is shown (y_{yellow}), indicating serious but manageable conditions under treatment.

When the system remains in this state for a specific duration range, it reveals a sensitive clinical pattern: the patient is experiencing a *critical condition that is responding to treatment* but requires continued intensive management. This temporal pattern could disclose the specific nature of the patient's medical condition and treatment response patterns, clinical decision-making processes and intervention strategies, and sensitive prognosis information that could be exploited by malicious actors.

We define the vulnerable states and secret dwell intervals as follows:

- Vulnerable states: $Q_v = \{(x_2, y_{\text{yellow}})\}$;
- Secret dwell interval: $\mathscr{I}_v((x_2, y_{\text{yellow}})) = [4\delta, 12\delta)$, where $\delta = 5$ minutes is the minimal dwell time.

The rationale for this interval is based on clinical significance. Very brief periods with duration $< 4\delta$ (less than 20 minutes) typically correspond to initial assessment and immediate intervention phases where the full clinical picture is still developing. The intermediate range $[4\delta, 12\delta)$ (20–60 minutes) specifically captures the sensitive pattern of a critical condition responding to treatment, revealing detailed information about the patient's specific condition, treatment efficacy, and clinical trajectory that must be protected from unauthorized access. Periods beyond 12δ (60 minutes or more) represent a different clinical scenario where sustained yellow alerts would typically trigger escalation to red alert or successful stabilization to green, and are thus excluded from the secret interval.

The timed secret is formally defined as:

$$S = \{((x_2, y_{\text{yellow}}), t) \mid t \in [4\delta, 12\delta)\}.$$

This secret captures all configurations where the patient monitoring system has determined the patient is in a Critical condition (state x_2) while displaying a yellow alert (output y_{yellow}) for a duration between 20 and 60 minutes. Timed opacity for this system means

that whenever the system is in a secret configuration, any observation sequence that could lead to this secret configuration could also be explained by some non-secret configuration. For instance, if the observer sees a continuous yellow alert lasting 35 minutes, this observation must remain consistent with either the system being in global state (x_1, y_{yellow}) for 35 minutes (non-secret) or in global state (x_2, y_{yellow}) for 35 minutes (which falls within the secret interval), making it impossible for the observer to infer the true sensitive clinical progression.

It is important to note that the minimal dwell time δ and secret interval bounds are deployment parameters that can be tailored to specific clinical protocols, system granularity, and privacy requirements. For instance, δ might range from 1–5 minutes depending on the monitoring system's temporal resolution and the clinical context.

6.2.4 Verification Methodology

Having defined the system model and timed secret specification, we now apply the verification framework from Chapter 4 to construct the evolution automaton, observer automaton, and perform opacity verification. The verification pipeline consists of three main stages: discretization into logical states, evolution automaton construction, and observer-based opacity checking.

6.2.4.1 Time Discretization and Logical States

The first step in the verification methodology is to discretize the continuous-time dwell semantics into a finite set of logical states. For each global state $q = (x, y) \in Q$, we compute the discretization parameter $\mathscr{R}(q)$ that determines the number of logical states needed to capture the timed behavior. For non-vulnerable states ($q \notin Q_v$), we have $\mathscr{R}(q) = 1$, yielding two logical states: q_0 (before minimal dwell time δ has elapsed) and q_1 (after δ has elapsed, with a self-loop for indefinite dwell). For vulnerable states with secret interval $[k'\delta, k''\delta)$, we have $\mathscr{R}(q) = k''$, yielding $k'' + 1$ logical states to track dwell time progression through and beyond the secret interval.

For the patient monitoring system, we compute $\mathscr{R}(q)$ for each global state:

Non-vulnerable global states:

- $q_1 = (x_0, y_{\text{green}})$: Since $q_1 \notin Q_v$, we have $\mathscr{R}(q_1) = 1$, yielding logical states $(x_0, y_{\text{green}})_0$ and $(x_0, y_{\text{green}})_1$.
- $q_2 = (x_1, y_{\text{yellow}})$: Similarly, $\mathscr{R}(q_2) = 1$, yielding $(x_1, y_{\text{yellow}})_0$ and $(x_1, y_{\text{yellow}})_1$.
- $q_3 = (x_2, y_{\text{red}})$: $\mathscr{R}(q_3) = 1$, yielding $(x_2, y_{\text{red}})_0$ and $(x_2, y_{\text{red}})_1$.

Vulnerable global state:

- $q_4 = (x_2, y_{\text{yellow}}) \in Q_v$ with secret interval $\mathscr{I}_v(q_4) = [4\delta, 12\delta)$. Here $k' = 4$ and $k'' = 12$, so $\mathscr{R}(q_4) = k'' = 12$.

  This yields 13 logical states:

$$(x_2, y_{\text{yellow}})_0, (x_2, y_{\text{yellow}})_1, \ldots, (x_2, y_{\text{yellow}})_{11}, (x_2, y_{\text{yellow}})_{12}$$

  Among these, the *secret logical states* are those with indices $j \in [k', k'') = [4, 12)$:

$$(x_2, y_{\text{yellow}})_4, (x_2, y_{\text{yellow}})_5, \ldots, (x_2, y_{\text{yellow}})_{11}$$

  These 8 logical states represent configurations where the patient has been in critical condition with yellow alert displayed for 20-60 minutes, revealing sensitive treatment response patterns that must be protected from unauthorized network access.

The time evolution for the vulnerable state progresses through the logical state sequence:

$$\begin{aligned}(x_2, y_{\text{yellow}})_0 &\xrightarrow{\delta} (x_2, y_{\text{yellow}})_1 \xrightarrow{\delta} \cdots \xrightarrow{\delta} (x_2, y_{\text{yellow}})_3 \\ &\xrightarrow{\delta} \underbrace{(x_2, y_{\text{yellow}})_4 \xrightarrow{\delta} \cdots \xrightarrow{\delta} (x_2, y_{\text{yellow}})_{11}}_{\text{secret logical states}} \\ &\xrightarrow{\delta} (x_2, y_{\text{yellow}})_{12} \xrightarrow{\delta} (x_2, y_{\text{yellow}})_{12} \xrightarrow{\delta} \cdots\end{aligned}$$

where the self-loop on $(x_2, y_{\text{yellow}})_{12}$ represents the system staying in this state for an indefinite period beyond $12\delta = 60$ minutes. In total, the system has 19 logical states: 6 non-vulnerable logical states (2 each for the three non-vulnerable global states) plus 13 logical states for the vulnerable state (x_2, y_{yellow}). Among these 19 logical states, exactly 8 are secret: $(x_2, y_{\text{yellow}})_4$ through $(x_2, y_{\text{yellow}})_{11}$.

6.2.4.2 Evolution Automaton Construction

Having established the logical state space, we now construct the evolution automaton $G_e(S)$ by applying Algorithm 4 from Chapter 4. The evolution automaton captures the full timed behavior of the system through four types of transitions labeled with symbols from the extended output alphabet $Y_e = Y \cup \{\delta, \varepsilon\}$: (1) δ-transitions representing time elapsing by one minimal dwell period; (2) Type 1 transitions labeled with ε representing unobservable discrete state changes without output changes; (3) Type 2 transitions

representing simultaneous discrete state and output changes; and (4) Type 3 transitions representing output changes without discrete state changes.

δ-transitions (time elapsing): From any logical state $(x,y)_j$ where $j < \mathscr{R}(x,y)$, time advances by one δ period to $(x,y)_{j+1}$. For example, $(x_0, y_{\text{green}})_0 \xrightarrow{\delta} (x_0, y_{\text{green}})_1$ represents a patient remaining stable for the first $\delta = 5$ minute period, and $(x_2, y_{\text{yellow}})_{10} \xrightarrow{\delta} (x_2, y_{\text{yellow}})_{11}$ represents a critical patient with yellow alert as time advances from 10δ to 11δ (50-55 minutes), still within the secret interval. When $j = \mathscr{R}(x,y)$, self-loops occur to represent indefinite dwell: $(x_0, y_{\text{green}})_1 \xrightarrow{\delta} (x_0, y_{\text{green}})_1$ indicates the stable state persists indefinitely, and $(x_2, y_{\text{yellow}})_{12} \xrightarrow{\delta} (x_2, y_{\text{yellow}})_{12}$ indicates the system remains beyond the secret window (indices 4–11), progressing indefinitely in this final logical state.

Type 1 transitions (ε-labeled): These unobservable transitions occur when the discrete state changes but the output remains the same, exploiting the nondeterministic output mapping. From any logical state $(x_1, y_{\text{yellow}})_j$ where $j > 0$, the system may transition to critical state x_2 without changing the yellow output: $(x_1, y_{\text{yellow}})_j \xrightarrow{\varepsilon} (x_2, y_{\text{yellow}})_0$ for any $j \geq 1$. This transition is unobservable since $y_{\text{yellow}} \in h(x_1) \cap h(x_2)$—the external observer sees continuous yellow alert but cannot detect that patient condition has worsened from "attention" to "critical". This creates the *observational confusion* crucial for opacity.

Type 2 (Simultaneous state and output change): These transitions simultaneously change both the discrete state and the output, making them observable. From $(x_0, y_{\text{green}})_j$ with $j > 0$, the patient may jump to the attention or critical state while the output switches to yellow: $(x_0, y_{\text{green}})_j \xrightarrow{y_{\text{yellow}}} (x_1, y_{\text{yellow}})_0$ and $(x_0, y_{\text{green}})_j \xrightarrow{y_{\text{yellow}}} (x_2, y_{\text{yellow}})_0$. The system may also transition directly to critical red alert: $(x_0, y_{\text{green}})_j \xrightarrow{y_{\text{red}}} (x_2, y_{\text{red}})_0$. Similarly, from attention or critical states, the system can return to stable green: $(x_1, y_{\text{yellow}})_j \xrightarrow{y_{\text{green}}} (x_0, y_{\text{green}})_0$ and $(x_2, y_{\text{yellow}})_j \xrightarrow{y_{\text{green}}} (x_0, y_{\text{green}})_0$.

Type 3 (Output change with no state change): These transitions change the output while remaining in the same discrete state. Inside the critical state, the output may toggle between yellow and red without any discrete-state change: $(x_2, y_{\text{yellow}})_j \xrightarrow{y_{\text{red}}} (x_2, y_{\text{red}})_0$ for $j > 0$. The observer sees the yellow-to-red switch but cannot definitively infer from that symbol alone whether a state change occurred, because multiple paths may lead to a red output.

Applying the algorithm to the patient monitoring system yields an evolution automaton with 19 logical states. The complete structure is illustrated in Figure 6.6, which

arranges the vulnerable state's 13 logical states vertically to clearly show the time progression, with bold δ-transitions marking entry to and exit from the secret region (shown in red shading). The figure demonstrates all four transition types and shows how the finite-state abstraction captures the complex timed behavior needed for verification.

6.2.4.3 Observer Automaton Construction

With the evolution automaton $G_e(S)$ constructed, we now build the observer automaton $\mathscr{O}(G_e)$ using Algorithm 5 from Chapter 4. The observer automaton computes the belief states (sets of logical states consistent with the observation sequence) using subset construction over the evolution automaton.

The observer construction requires three key components. First, the *observation abstraction function* ψ maps continuous-time output observations to discrete observation sequences over $Y \cup \{\delta\}$. For example, if an observer records a green alert for 7 minutes followed by a yellow alert for 23 minutes with $\delta = 5$ minutes, the abstraction yields $\psi((y_{\text{green}}, 7)) = \delta$ and $\psi((y_{\text{green}}, 7)(y_{\text{yellow}}, 23)) = \delta\, y_{\text{yellow}}\, \delta^4$, where fractional time units are discarded (the 7-minute green alert becomes one δ-step, and the 23-minute yellow alert becomes four δ-steps). This discretization preserves the key timing boundaries defined by the secret intervals.

Second, the *reachability computation* determines which logical states are reachable from a given belief state under a specific observation symbol. For an observable output symbol $y \in Y$, we compute $\alpha(z,y)$ as the set of logical states reachable via y-labeled transitions from any state in belief state z. We then compute $\beta(z,y) = D_\varepsilon(\alpha(z,y))$, the ε-closure of $\alpha(z,y)$, to account for unobservable transitions that may follow the observable transition. For example, consider the transitions $(x_0, y_{\text{green}})_1 \xrightarrow{y_{\text{yellow}}} (x_1, y_{\text{yellow}})_0$, $(x_0, y_{\text{green}})_1 \xrightarrow{y_{\text{yellow}}} (x_2, y_{\text{yellow}})_0$, and $(x_1, y_{\text{yellow}})_0 \xrightarrow{\varepsilon} (x_2, y_{\text{yellow}})_0$. From belief state $z = \{(x_0, y_{\text{green}})_1\}$, observing y_{yellow} yields $\alpha(z, y_{\text{yellow}}) = \{(x_1, y_{\text{yellow}})_0, (x_2, y_{\text{yellow}})_0\}$, and computing the ε-closure gives $\beta(z, y_{\text{yellow}}) = \{(x_1, y_{\text{yellow}})_0, (x_2, y_{\text{yellow}})_0\}$ since the unobservable transition from $(x_1, y_{\text{yellow}})_0$ to $(x_2, y_{\text{yellow}})_0$ is already included. This belief state captures the critical observational uncertainty: the observer cannot distinguish whether the patient is in attention state or critical state—both are consistent with the yellow alert observation.

Third, the *subset construction* initializes the observer with $z_0 = D_\varepsilon((x_0, y_{\text{green}})_0)$ and explores all reachable belief states by iteratively computing $\beta(z,y)$ for each belief state z and observation symbol $y \in Y \cup \{\delta\}$. The algorithm terminates when no new belief states are discovered.

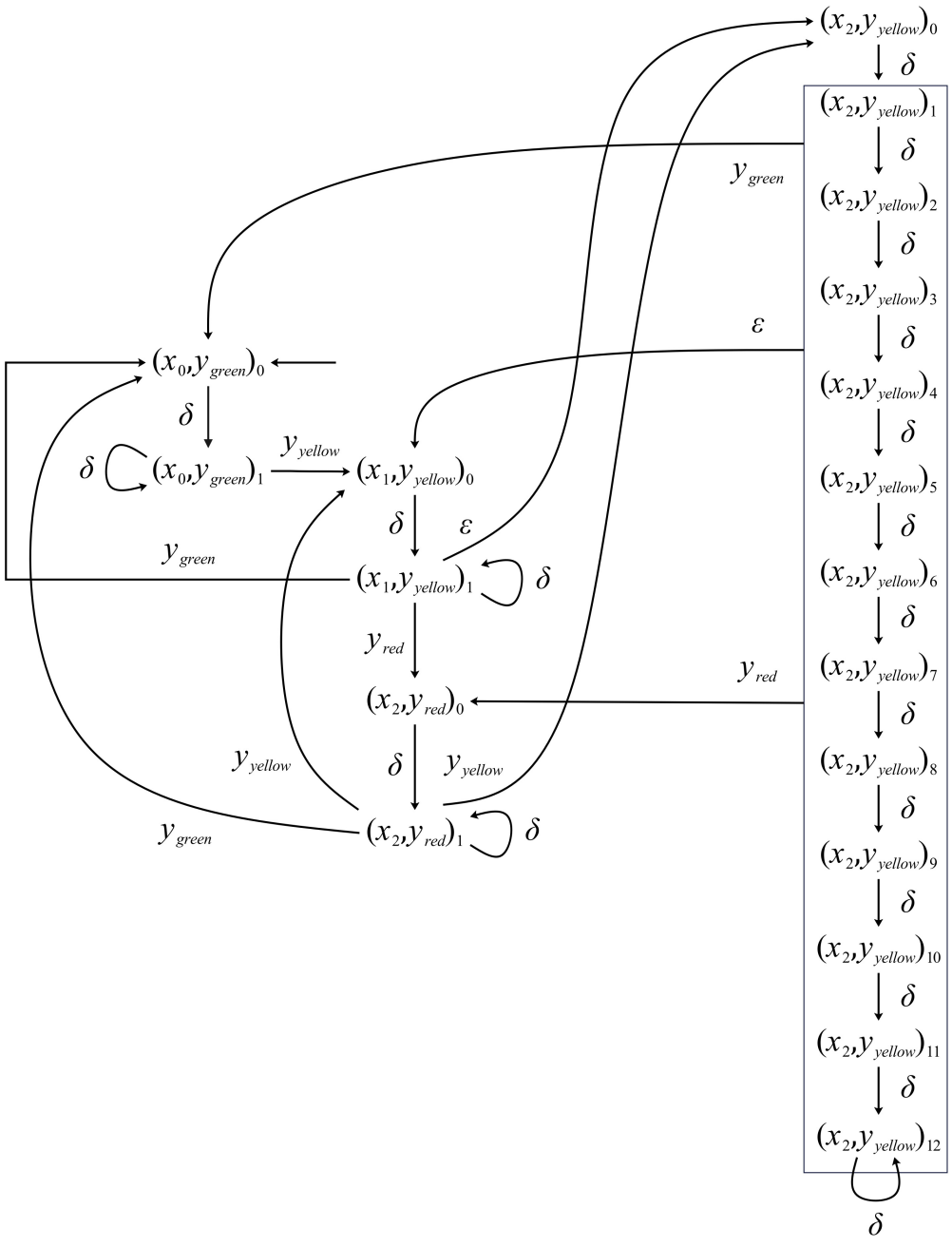


Figure 6.6: Evolution automaton $G_e(S)$ for the patient monitoring system.

For the patient monitoring system, the observer construction proceeds as follows. From the initial belief state $z_0 = \{(x_0, y_{\text{green}})_0\}$, a δ-transition yields $z_1 = \{(x_0, y_{\text{green}})_1\}$. From z_1, a y_{yellow} observation yields $z_2 = \{(x_1, y_{\text{yellow}})_0, (x_2, y_{\text{yellow}})_0\}$ due to the nondeterministic transitions and ε-closure.

As δ-transitions advance time, the belief states expand to include progressively more logical states of the vulnerable state (x_2, y_{yellow}), while always maintaining the non-vulnerable attention state (x_1, y_{yellow}) logical states. This expansion reflects the observational uncertainty: yellow alert observations are consistent with both attention-level and critical-level conditions. For instance, after several δ-transitions from z_2, the observer reaches states z_5 (containing logical states up to $(x_2, y_{\text{yellow}})_8$) and z_7 (containing all logical states up to $(x_2, y_{\text{yellow}})_{12}$). The detailed composition of all observer states is provided in Table 6.7.

The complete observer automaton construction yields 17 reachable belief states, as shown in Figure 6.7. Table 6.7 provides the detailed composition of each observer state, explicitly listing which logical states belong to each belief state.

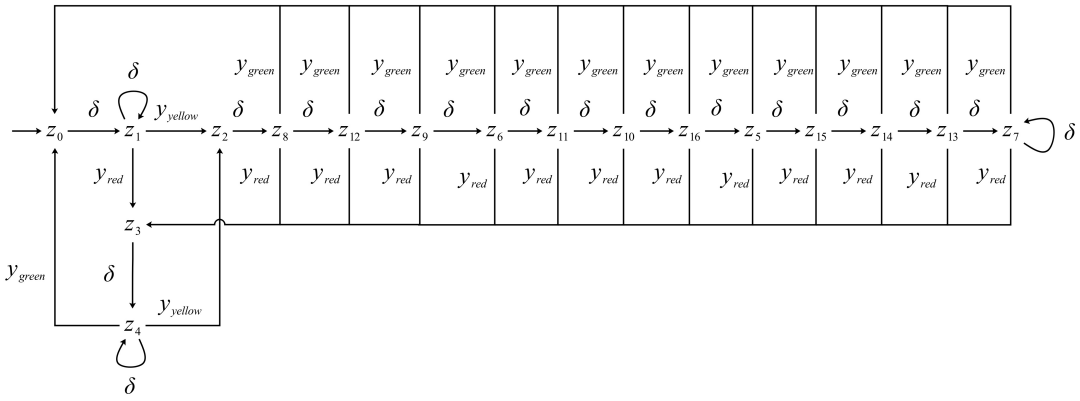


Figure 6.7: Complete observer automaton for the patient monitoring system.

Table 6.7: Observer state composition for the patient monitoring system.

Observer State	**Logical State Set**
z_0	$(x_0, y_{\text{green}})_0$
z_1	$(x_0, y_{\text{green}})_1$
z_2	$(x_1, y_{\text{yellow}})_0, (x_2, y_{\text{yellow}})_0$
z_3	$(x_2, y_{\text{red}})_0$
z_4	$(x_2, y_{\text{red}})_1$

(Continued on next page)

(Continued from previous page)

Observer State	**Logical State Set**
z_5	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_8$
z_6	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_4$
z_7	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_{12}$
z_8	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, (x_2, y_{\text{yellow}})_1$
z_9	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_3$
z_{10}	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_6$
z_{11}	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_5$
z_{12}	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_2$
z_{13}	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_{11}$
z_{14}	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_{10}$
z_{15}	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_9$
z_{16}	$(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_7$

6.2.5 Opacity Verification

With both the evolution automaton and observer automaton constructed, we can now perform the opacity verification using Theorem 4.1 from Chapter 4. Recall that the secret logical state set is

$$S_v = \{(x_2, y_{\text{yellow}})_4, (x_2, y_{\text{yellow}})_5, \ldots, (x_2, y_{\text{yellow}})_{11}\},$$

corresponding to dwell times in the interval $[4\delta, 12\delta) = [20, 60)$ minutes in critical condition with yellow alert.

The opacity verification algorithm checks whether any reachable observer state z satisfies $z \subseteq S_v$—that is, whether any belief state contains exclusively secret logical states. If such a state exists, the observer can conclusively infer that the system is in a secret configuration, violating timed opacity. Conversely, if no such state exists, the system is timed opaque: every observation sequence leading to a secret configuration is also consistent with at least one non-secret configuration.

Examining the observer states in Table 6.7, we observe that every reachable observer state containing yellow outputs includes logical states from *both* the attention state x_1 and the critical state x_2. Crucially, these belief states also include logical states *outside* the secret interval:

- $z_2 = \{(x_1, y_{\text{yellow}})_0, (x_2, y_{\text{yellow}})_0\}$ contains the non-secret state $(x_2, y_{\text{yellow}})_0$ (dwell time 0, before the secret interval);
- $z_5 = \{(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_8\}$ contains non-secret attention states (indices 0–1) and non-secret critical states (indices 0–3, corresponding to dwell times 0–15 minutes before the secret interval);
- $z_7 = \{(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_{12}\}$ contains non-secret attention states and non-secret critical states (indices 0–3 and 12, corresponding to dwell times 0–15 minutes and 60+ minutes outside the secret interval);
- $z_{13} = \{(x_1, y_{\text{yellow}})_0, (x_1, y_{\text{yellow}})_1, (x_2, y_{\text{yellow}})_0, \ldots, (x_2, y_{\text{yellow}})_{11}\}$ contains all secret logical states plus non-secret attention states and early critical states (indices 0–3).

Consequently, no reachable observer state satisfies $z \subseteq S_v$. By Theorem 4.1, the patient monitoring system is **timed opaque** with respect to the timed secret $S = \{((x_2, y_{\text{yellow}}), t) \mid t \in [4\delta, 12\delta)\}$.

6.3 Battery Management System: Timed Fault Diagnosis

In this section, we present a comprehensive case study that demonstrates the complete application of the SOAF framework to a real-world safety-critical system: an electric vehicle battery management system. This case study integrates all the concepts introduced in Chapter 5—fault-prone states, fault arcs, fault time mappings, the Evolution Automaton with Faults, and the diagnoser construction—into a unified analysis of time-dependent thermal runaway diagnosis.

6.3.1 System Background and Motivation

Modern electric vehicles rely on sophisticated battery management systems to monitor and control lithium-ion battery packs during operation [116]. The BMS continuously tracks battery state variables (voltage, current, temperature, state of charge) and coordinates charging/discharging operations to ensure safe and efficient operation [116].

The thermal runaway hazard. Among the safety concerns that BMS must address, the most critical is *thermal runaway*—a catastrophic failure mode where exothermic chemical reactions within a battery cell generate heat faster than it can be dissipated [117]–[119].

If unchecked, thermal runaway leads to cell rupture, fire, and potential propagation to neighboring cells, causing cascading failures across the entire battery pack [117]–[119].

The key characteristic of thermal runaway is its *time-dependent* nature: it develops through a cumulative multi-stage degradation process. Heat accumulation accelerates chemical side reactions (e.g., SEI decomposition, electrolyte breakdown, lithium plating), with reaction rates increasing strongly with temperature (Arrhenius-type dependence) [117]–[119]. Experimental data shows strong time–heating coupling: for instance, under imposed external heating, increasing heat flux from 1.5 kW/m^2 to 2.0 kW/m^2 reduced the time to thermal runaway from beyond $\sim$ 20 minutes to about $\sim$ 15–16 minutes in one study [120]. Once internal temperatures rise into the regime where key components (SEI, separator, electrolyte, cathode) undergo accelerated exothermic decomposition, catastrophic failure can progress rapidly and becomes difficult to arrest [117]–[119]. The BMS must detect precursor conditions before this point [121], [122].

Observable indicators and diagnosis challenge. The BMS cannot directly observe internal cell degradation, and surface measurements can lag internal structural damage [121]. Instead, the BMS relies on indirect measurements: thermistors classify thermal states (safe, warm, hot, critical); voltage and current sensors detect anomalies; and state estimation algorithms (EKF/UKF-type filters) infer SOC and SOH deviations [116], [121], [123].

The diagnostic challenge is to *infer the risk of imminent thermal runaway* from these external observations combined with timing information [121], [122]. Crucially, the relationship between observable surface temperature (and other external signals) and internal failure risk is *time-dependent and nonlinear*, and the time-to-runaway can vary substantially across abuse intensities and operating conditions [120]–[122]. Moreover, distinct failure mechanisms operate over different temporal windows: *acute failures* (minutes) associated with internal short circuits, separator shrinkage, or dendrite penetration, and *chronic failures* (tens of minutes to hours) via cumulative electrolyte decomposition and SEI degradation [117]–[119], [124], [125].

To address this diagnostic challenge, we abstract BMS operation into three discrete modes characterized by discharge current profiles:

- **Inactive mode:** Zero discharge current (vehicle parked or emergency shutdown). The BMS enters a low-power monitoring state [116]. Due to thermal inertia, battery temperature may remain elevated immediately after shutdown and gradually cools as heat dissipates through the thermal management system [116], [118].

- **Normal discharge mode:** Moderate discharge rates during typical driving conditions (city traffic, highway cruising). Cell temperatures typically remain within safe or warm ranges, and the BMS performs routine monitoring and state estimation [116], [123], [126].

- **Fast discharge mode:** High discharge rates under peak power demand (rapid acceleration, hill climbing, high-speed driving). This imposes significant thermal and electrical stress on cells, increasing the likelihood of fault precursors that can evolve toward thermal runaway if residence time in high-stress conditions is long enough [117]–[119], [121], [122], [124].

While hybrid/model-based approaches for battery fault diagnosis have modeled multiple fault modes and employed observer-based estimation frameworks [123], [124], [126], we propose a simplified three-state abstraction that captures essential thermal dynamics arising from different discharge current levels while maintaining computational tractability for formal verification. This abstraction enables finite-state diagnosis of residence-time-triggered thermal runaway faults—the focus of our framework—without requiring detailed modeling of continuous electrochemical dynamics. Unlike hybrid approaches that require solving continuous differential equations at runtime, a finite-state (timed/discrete-event) model supports offline verification and diagnosability analysis with well-studied decidability properties [10], [12].

Modeling requirements and hybrid system representation. To effectively diagnose thermal runaway risk, we require a modeling framework that captures both continuous dynamics (temperature evolution, voltage/current profiles) and discrete events (mode transitions, fault occurrences) [127], [128]. Recent research in battery fault diagnosis has explored hybrid system approaches, including stochastic hybrid automata and hybrid-state estimation frameworks, which model battery systems by integrating continuous state variables (SOC, voltage, temperature) with discrete fault modes (normal, sensor fault, thermal fault) [124], [129]–[131]. In addition, Petri net / hybrid Petri net formalisms have been used to model and verify safety-critical energy systems that contain battery energy storage, supporting discrete-event abstraction and formal analysis [132], [133]. These hybrid/discrete-event models avoid relying solely on residuals and thresholds for fault detection, enabling more robust multi-fault isolation across different levels of aggregation [124], [127], [134], [135].

Building on this foundation, the SOAF framework we develop must capture:

1. **Discrete operational modes:** The system operates in three fundamental states—inactive, normal discharge, fast discharge—corresponding to distinct discharge current levels and thermal profiles [127], [130], [131].

2. **Observable outputs evolving within modes:** Temperature classifications (safe, warm, hot) that change over time *without discrete state transitions*—a key distinguishing feature of the SOA framework that naturally represents continuous temperature rise during sustained operation in a fixed discharge mode [128].

3. **Time-gated fault transitions with output dependency:** Thermal runaway faults can only occur after dwelling in fast discharge mode for specific durations, with critical time thresholds depending on the observable temperature (experimental evidence shows strong coupling between heating severity and time-to-thermal-runaway) [119], [120].

4. **Multiple vulnerability windows for heterogeneous failure modes:** Capturing both acute failures (separator breach within $5-10$ minutes) and chronic failures (electrolyte decomposition over $20-30$ minutes) through disjoint time intervals $[\delta_1', \delta_1''), [\delta_2', \delta_2'')$ for each fault-prone global state, reflecting distinct failure mechanisms operating on different timescales [119], [124], [127].

5. **Emergency shutdown as anomalous transition:** When thermal runaway is imminent, the BMS forces an emergency transition directly from fast discharge (x_2) to inactive mode (x_0), bypassing normal discharge (x_1). This fault arc $(x_2, x_0) \in B_f$ is *disjoint* from nominal behavior B, making it diagnosable through transition structure alone, without requiring special fault-indicating outputs [124], [127].

6.3.2 SOAF Model Construction

Based on the requirements above, we construct the BMS SOAF with four key components: (1) three-state abstraction (x_0, x_1, x_2) based on discharge current; (2) composite safety assessments $(y_{\text{safe}}, y_{\text{caution}}, y_{\text{critical}})$ fusing multiple sensor modalities; (3) single fault arc (x_2, x_0) representing emergency shutdown; and (4) output-conditioned fault time windows capturing temperature-dependent vulnerability with dual acute/chronic failure mechanisms.

Example 6.1 (BMS SOAF Model)**.** *We now formalize the complete SOAF model, illustrated in Figure 6.8.*

Base SOA. *The BMS operation is abstracted into three discrete states based on discharge current magnitude:*

- x_0 ***(Inactive mode):*** *Zero discharge current (vehicle parked or emergency shutdown). Although current is zero, battery temperature may remain elevated (e.g.,* $50^\circ C$*) immediately after shutdown due to thermal inertia. Residual heat from prior high-power operation dissipates passively through the cooling system over 5-10 minutes.*

- x_1 ***(Normal discharge mode):*** *Moderate discharge rates during typical driving conditions (city traffic, highway cruising). Cell temperatures typically remain within safe ranges (*$< 45^\circ C$*), and the BMS performs routine monitoring without triggering elevated risk assessments.*

- x_2 ***(Fast discharge mode):*** *High discharge rates under peak power demand (rapid acceleration, hill climbing, high-speed driving). This imposes significant thermal and electrical stress on cells, with temperatures potentially reaching* $45-55^\circ C$ *or higher during sustained operation. Only this mode poses thermal runaway risk under prolonged dwell.*

Formally, $G_{BMS} = (X, Y, B, h, x_0, y_0)$ *with state set* $X = \{x_0, x_1, x_2\}$*, output alphabet* $Y = \{y_{safe}, y_{caution}, y_{critical}\}$ *representing composite safety assessments, nominal arcs* $B = \{(x_0, x_1), (x_1, x_2), (x_2, x_1), (x_1, x_0)\}$ *representing normal operational transitions:*

- (x_0, x_1)*: starting vehicle (inactive* $\rightarrow$ *normal discharge)*

- (x_1, x_2)*: increasing power demand (normal* $\rightarrow$ *fast discharge)*

- (x_2, x_1)*: reducing power demand (fast* $\rightarrow$ *normal discharge)*

- (x_1, x_0)*: stopping vehicle (normal discharge* $\rightarrow$ *inactive)*

The output mapping is:

$h(x_0) = \{y_{safe}\}$ *(inactive mode always safe:* $I = 0A$*, no heat generation),*
$h(x_1) = \{y_{safe}, y_{caution}\}$ *(normal discharge may reach safe or caution levels),*
$h(x_2) = \{y_{safe}, y_{caution}, y_{critical}\}$ *(fast discharge can reach any risk level)*

A critical design aspect: these outputs are not *pure temperature readings but* composite safety assessments *that fuse multiple sensor modalities:*

- ***Temperature*** *(primary indicator):* $< 35°C \rightarrow$ *safe;* $35-45°C \rightarrow$ *caution;* $45-55°C \rightarrow$ *critical, following recommended operating/safety temperature ranges and thermal-safety discussions [118], [136].*

- ***Discharge current****: Zero current* ($I = 0A$) *implies no heat generation* ($\dot{Q}_{gen} = I^2R \approx 0$)*, so even high temperature is safe (residual heat dissipates passively).*

- ***Voltage anomalies****: Sudden voltage drops* ($> 10\%$) *or abnormal voltage behavior can indicate internal short-related hazards or separator/dendrite issues [137]–[139].*

- ***State estimation****: SOC/SOH deviations inferred by EKF/UKF (sigma-point Kalman) algorithms signal internal degradation that amplifies thermal risk [123], [126].*

This multi-source fusion enables context-aware risk assessment:

- ***Example 1:*** *After emergency shutdown,* $T = 50°C$ + $I = 0A \rightarrow y_{safe}$*. Despite elevated temperature, zero current eliminates further heat generation, making the condition safe (thermal decay after shutdown follows electro-thermal dynamics on the order of minutes [140]). This is why* $h(x_0) = \{y_{safe}\}$ *is a singleton.*

- ***Example 2:*** *During moderate discharge,* $T = 40°C$ *(nominally caution-level) +* 15% *voltage drop* $\rightarrow y_{critical}$*. The voltage anomaly signals potential internal short circuit capable of producing internal Joule heating and a measurable internal temperature jump [139], overriding the moderate surface temperature and necessitating critical assessment.*

The system starts in configuration (x_0, y_{safe}) *with minimal dwell time* $\delta = 10$ *seconds.*

Fault structure. *We extend the base SOA with fault arc* $B_f = \{(x_2, x_0)\}$ *representing emergency thermal shutdown. When thermal runaway is imminent, the BMS executes an emergency jump* $x_2 \rightarrow x_0$ *(fast discharge to inactive), bypassing the nominal path* $x_2 \rightarrow x_1 \rightarrow x_0$*. Since* $(x_2, x_0) \notin B$ *but* $(x_2, x_0) \in B_f$*, this anomalous transition becomes diagnostic evidence for fault occurrence—the SOAF framework can identify faults through transition structure alone, without requiring special fault-indicating outputs.*

This yields fault-prone state $X_f = \{x_2\}$*, as only fast discharge poses thermal runaway risk under prolonged dwell, resulting in fault-prone global states*

$$Q_f = \{(x_2, y_{safe}), (x_2, y_{caution}), (x_2, y_{critical})\}.$$

Fault time mapping. *The temporal constraints capture temperature-dependent thermal runaway risk (with $\delta = 10$ seconds, for nominal battery health), based on experimental thermal-runaway characterization and internal-short evolution/staging, and known aging effects on TR behavior [137], [139], [141], [142]:*

$$\mathscr{I}((x_2, y_{safe}), (x_2, x_0)) = \{[30\,min, +\infty)\}$$
$$\mathscr{I}((x_2, y_{caution}), (x_2, x_0)) = \{[15\,min, +\infty)\}$$
$$\mathscr{I}((x_2, y_{critical}), (x_2, x_0)) = \{[5\,min, 10\,min), [20\,min, +\infty)\}$$

These time windows reflect temperature-dependent degradation mechanisms:

- ***Safe temperature regime*** (x_2, y_{safe})***:*** $[30\,\mathbf{min}, +\infty)$. *At safe surface temperatures ($< 35°C$) during fast discharge, cumulative stress from prolonged high current operation (volume expansion/contraction, SEI thickening, interfacial degradation) accumulates over tens of minutes before reaching critical thresholds. The 30-minute window captures the time required for cumulative damage to trigger thermal runaway when sustained at elevated power levels.*

- ***Caution temperature regime*** $(x_2, y_{caution})$***:*** $[15\,\mathbf{min}, +\infty)$. *At elevated temperatures ($35-45°C$), reaction/aging kinetics accelerate with Arrhenius-type temperature dependence [143], [144]. Accelerated SEI decomposition, electrolyte breakdown, and reduced thermal margin to separator melting enable thermal runaway onset within 15-20 minutes under sustained operation.*

- ***Critical temperature regime*** $(x_2, y_{critical})$***:*** $[5, 10) \cup [20, +\infty)$. *At critical temperatures ($45-55°C$), thermal runaway mechanisms bifurcate into two distinct paths:*

  - ***Acute failure*** $[5, 10)$ ***minutes:*** *Rapid dendrite penetration / separator failure and internal short formation are widely recognized triggering mechanisms [137], [138], [145].*

  - ***Thermal stabilization window*** $[10, 20)$ ***minutes:*** *Active cooling systems may temporarily stabilize temperature if discharge rate decreases, though cumulative chemical damage continues.*

  - ***Chronic failure*** $[20, +\infty)$ ***minutes:*** *Cumulative gas generation (CO_2, C_2H_4), SEI decomposition/regrowth cycles, transition metal dissolution from cathode, and PVDF binder decomposition lead to thermal runaway through slow chemical degradation pathways.*

The disjoint interval representation $\{[5,10),[20,+\infty)\}$ *captures this dual-mechanism structure: thermal runaway can occur via fast acute paths or slow chronic paths, with temporary stabilization possible in the intermediate window.*

Health-dependent parameterization. *For batteries with degraded State of Health (*$\theta < 1$*, where* $\theta = 1$ *represents new battery and* $\theta \approx 0.8$ *represents end-of-life by industry standards [146]), these time windows scale by factor* θ*. For example:*

$$\mathscr{I}_\theta((x_2, y_{critical}),(x_2,x_0)) = \{[\theta \cdot 5\,min, \theta \cdot 10\,min),[\theta \cdot 20\,min,+\infty)\}.$$

An aging battery with $\theta = 0.8$ *exhibits acute failure window* $[4,8)$ *minutes instead of* $[5,10)$*, reflecting reduced thermal tolerance (thinner separators, increased internal resistance, compromised cooling efficiency). In subsequent analysis, we assume nominal health* $\theta = 1$ *for clarity.*

Remark 6.1 (Set-Valued Output Mapping and Diagnosis)**.** *The output mapping exhibits a nested structure* $h(x_0) \subset h(x_1) \subset h(x_2)$*, reflecting the physical constraint that higher discharge currents expand the range of possible thermal states (*$\dot{Q}_{gen} \propto I^2$*). This design enables two key features for SOAF diagnosis:*

Within-mode output evolution: *During sustained operation in state* x_2*, the system can traverse the global state sequence* $(x_2, y_{safe}) \to (x_2, y_{caution}) \to (x_2, y_{critical})$ *as temperature rises continuously, without requiring explicit state transitions. The set-valued mapping thus provides a discrete abstraction of continuous temperature dynamics while maintaining a finite state space—this is the distinguishing feature of SOA that enables time-dependent fault diagnosis with finite-state techniques.*

Partial state identification for diagnosis: *Observing* $y_{critical}$ *uniquely identifies state* x_2*, while* y_{safe} *is ambiguous. However, the singleton property of* $h(x_0)$ *enables exclusivity reasoning: if the system transitions from* $y_{critical}$ *to* y_{safe} *in one step and* y_{safe} *persists, this suggests entry into* x_0*. Combined with transition structure constraints—*$(x_2,x_0) \notin B$ *but* $(x_2,x_0) \in B_f$*—this output pattern becomes diagnostic evidence for fault occurrence, as demonstrated in Section 6.3.3.*

Figure 6.9 visualizes the fault time windows across the three temperature regimes, illustrating how higher temperatures progressively reduce safe operating duration and enable distinct failure mechanisms (acute vs. chronic) at critical-risk levels.

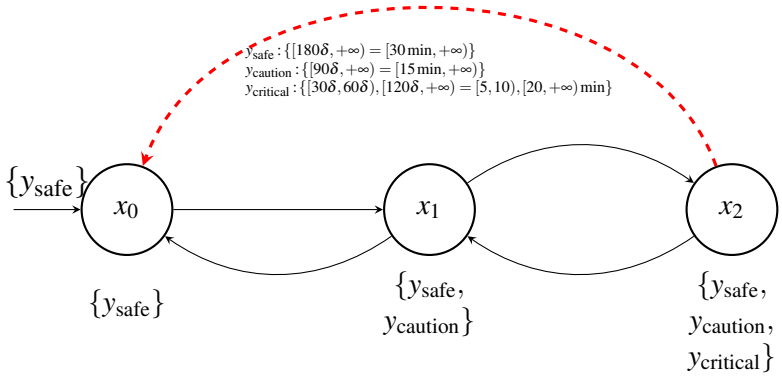
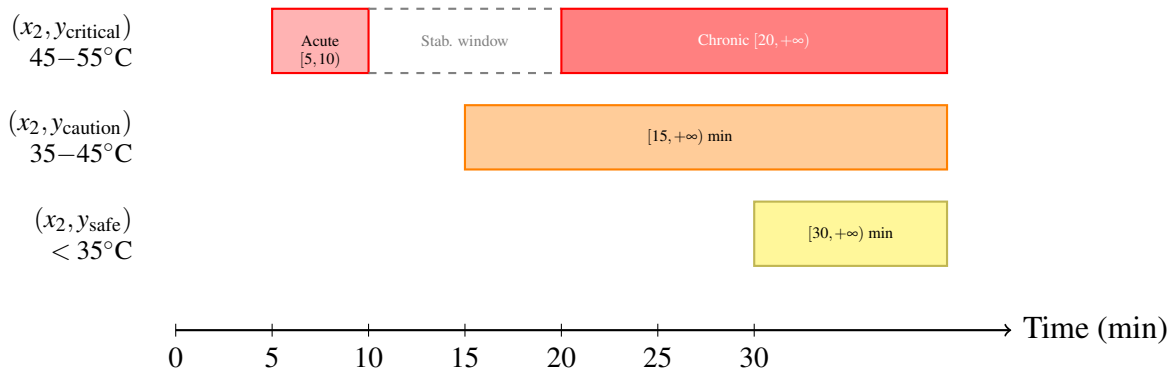


Figure 6.8: BMS SOAF with initial state x_0 and minimal dwell time $\delta = 10$s.



Shaded regions: fault arc (x_2, x_0) enabled.
Darker shading indicates higher criticality.

Figure 6.9: Fault time windows for BMS thermal runaway diagnosis.

6.3.3 Diagnostic Scenarios

We now demonstrate the complete four-stage diagnosis pipeline (Section 5.3) applied to the BMS SOAF model. Applying the construction algorithms from Sections 5.3.1–5.3.4 to the BMS SOAF (Example 6.1) yields the automata specifications summarized in Table 6.8.

Table 6.8: Diagnostic pipeline summary: state-space dimensions for the BMS case study.

	Evolution Automaton with Faults	**Fault Recognizer**	**Diagnoser**
States	399	798	4388
Transitions	2004	4008	3724

Due to the large state spaces, we provide partial visualizations of the key diagnostic structures in Figures 6.10–6.12 to illustrate the construction pipeline. We then demonstrate the diagnostic function through observation trace evaluation.

Consider an aggressive highway driving scenario with observation sequence:

$$\omega = (y_{\text{safe}}, 50\delta) \cdot (y_{\text{caution}}, 80\delta) \cdot (y_{\text{critical}}, 40\delta) \cdot (y_{\text{safe}}, 10\delta) \tag{6.6}$$

This corresponds to: safe for 8.3 min, caution for 13.3 min, critical for 6.7 min, then safe for 1.7 min. Processing ω through the diagnoser yields:

$$\varphi(z_{\text{final}}) = U \quad \text{(UNCERTAIN)}$$

6.4 Summary

The three case studies presented in this chapter validate the SOA framework's applicability to diverse cyber-physical systems with distinct verification objectives:

- **Smart water supply system (Section 6.1).** The current-state opacity verification confirmed that aggregate power measurements preserve confidentiality of critical operational states. Despite an adversary observing the complete power consumption history, the inherent observational ambiguity—multiple distinct actuator configurations yielding identical power levels—prevents conclusive inference of vulnerable tank conditions. The 20-state SOA abstraction with 9 discrete power levels, combined with $\delta = 3$ s discretization, yielded a 64-state evolution automaton and 52-state observer with 199 transitions, demonstrating computational tractability for verification.

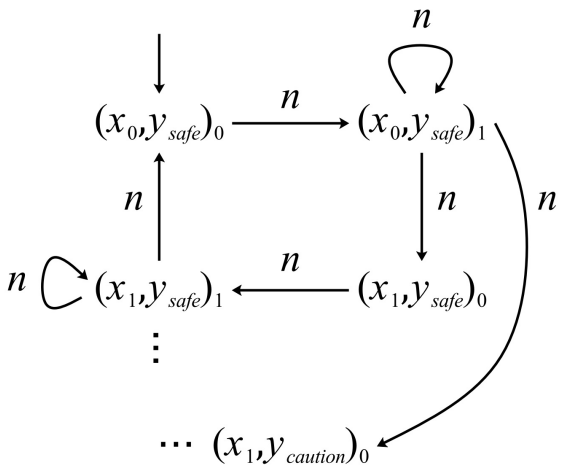


Figure 6.10: Evolution Automaton with Faults for the BMS case study (partial view showing representative states and transitions).

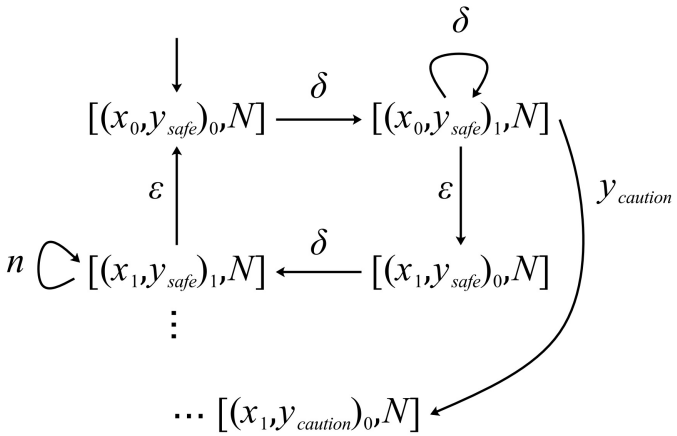


Figure 6.11: Fault Recognizer for the BMS case study (partial view illustrating the composition with the fault monitor).

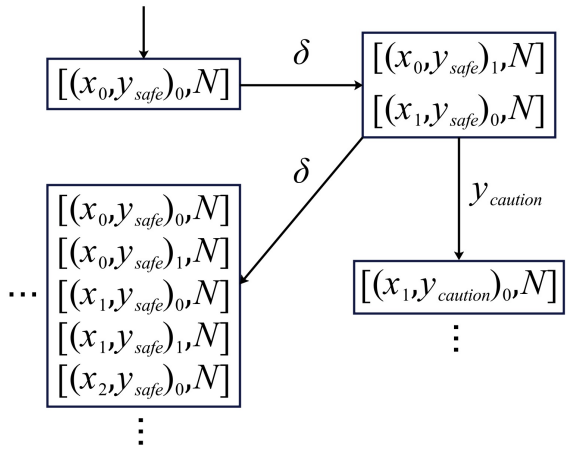


Figure 6.12: Diagnoser for the BMS case study (partial view showing belief state structure and diagnostic labels).

- **Patient monitoring system (Section 6.2).** The timed opacity verification demonstrated protection of sensitive clinical information through observational ambiguity. The 3-state SOA with 3 alert levels yielded a 19-state evolution automaton and 17-state observer, confirming that yellow alert observations lasting 20–60 minutes cannot conclusively reveal critical patient conditions. This illustrates the framework's applicability to healthcare privacy protection.

- **Battery management system (Section 6.3).** The timed fault diagnosis framework successfully detected thermal runaway faults triggered by dwelling in high-temperature discharge modes. The SOAF model with 3 operational states and 3 composite safety assessments, combined with output-conditioned fault windows capturing temperature-dependent vulnerability, yielded a 399-state evolution automaton with faults and a 4388-state diagnoser. The case study illustrated how health-dependent parameterization and multi-source observables reduce detection delay while maintaining diagnostic certainty.

All three case studies highlight the framework's key strengths: (i) alignment with physical observability constraints (aggregate measurements, sensor fusion, alert levels), (ii) decidable verification through finite-state abstraction, and (iii) scalability to realistic system dimensions without resorting to conservative approximations or symbolic representations. The output-with-duration modeling paradigm proves both mathematically tractable and practically faithful to field deployment scenarios across critical infrastructure, healthcare, and energy storage systems.

Implementation and reproducibility. To facilitate reproducibility and enable further research, we have developed SOA Toolbox, an open-source implementation of the verification and diagnosis framework, publicly available on GitHub.[1] The toolbox is implemented in Python and MATLAB, providing command-line interfaces for model specification, algorithm execution, and result analysis. It implements the core algorithms for evolution automaton construction, observer synthesis, opacity verification, and fault diagnosis presented in Chapters 3–5. All three case studies presented in this chapter can be reproduced using the provided implementation, with model specifications and verification scripts included in the repository. This open-source contribution aims to lower the barrier to applying formal timing-aware verification methods in cyber-physical systems and to support the research community in extending the framework to additional application domains.

[1] https://github.com/ty451/Switching-Output-Automata

Chapter 7

Conclusion and Future Work

7.1 Conclusions

This dissertation developed a finite-state approach to timing-aware security and reliability analysis for cyber-physical systems by introducing the Switching Output Automaton (SOA) framework. The core idea is to reason directly on *outputs and how long they persist*, aligning formal models with what observers actually measure in practice. Chapter 3 formalized SOA as a continuous-time discrete-event model combining set-valued output mappings (capturing measurement aggregation), output switching within states (reflecting piecewise-constant sensor readings), and minimal dwell time constraints (bounding observable change frequency). These features enable finite-state abstraction: discretizing each global state into logical states tracking dwell-time progression yields evolution automata with finitely many states despite continuous-time semantics. Deterministic state estimation through observer construction reduces verification to reachability analysis, achieving decidability without the syntactic restrictions or zone-based symbolic computation required for timed automata.

Within this framework, the dissertation addresses two complementary verification problems:

- **Timed opacity** (Chapter 4). We formalized secrets as global-state-duration triples specifying which combinations of discrete state, output, and dwell time must remain confidential, capturing properties where dwelling duration is as sensitive as state identity. The key contribution is secret-dependent time discretization: applying fine-grained quantization only to vulnerable global states yields finite evolution automata whose size scales with secret complexity rather than uniform discretization. Verification reduces to checking whether any reachable observer state consists

entirely of secret logical states. This restores decidability for timed opacity without restricting to timed automata subclasses (real-time automata, constant-time labeled automata) or weakening intruder capabilities, keeping complexity within the observer-based determinization profile familiar from untimed discrete-event systems.

- **Timed fault diagnosis** (Chapter 5). We extended SOA to the Switching Output Automaton with Faults (SOAF) by partitioning transitions into nominal arcs (always enabled after minimal dwell time) and time-gated fault arcs that activate only when dwell time in a global state falls within specified enablement windows, modeling degradation from sustained stress exposure (battery thermal runaway, semiconductor junction failure, structural fatigue). A four-stage pipeline—logical state discretization, Evolution Automaton with Faults construction, fault recognizer synthesis, and subset determinization—yields a deterministic online diagnoser classifying observation sequences as NORMAL, FAULTY, or UNCERTAIN with sound/complete detection and bounded delay guarantees. Language preservation theorems ensure the finite-state abstraction captures all SOAF behaviors up to time quantization without loss of diagnostic information.

Chapter 6 demonstrated practical applicability through three case studies. The DTS200 smart water supply system verified current-state opacity under aggregate power monitoring, confirming that discretized power consumption measurements with appropriate minimal dwell time provide sufficient observational ambiguity to protect critical pump-valve configurations. The patient monitoring system verified timed opacity for healthcare privacy, ensuring prolonged critical-condition patterns with yellow alerts remain indistinguishable from benign fluctuations through external network observations. The battery management system applied timed fault diagnosis to thermal runaway detection, illustrating how health-dependent fault window parameterization and multi-source observables reduce detection delay while maintaining diagnostic certainty. To support reproducibility and enable further research, SOA Toolbox has been released as an open-source implementation in Python and MATLAB, providing command-line tools for evolution automaton construction, observer synthesis, opacity verification, fault diagnosis, and reproduction of all case studies presented.

By modeling cyber-physical systems at the level of observable outputs and their durations rather than internal states or dense-time dynamics, the SOA framework achieves decidable verification of timing-dependent security and reliability properties. The minimal dwell time constraint discretizes continuous time into finite logical structures amenable to reachability analysis, handling realistic system scales while providing formal guarantees

absent from simulation-based approaches. The framework bridges theoretical decidability and practical deployment by aligning formal models with field observability constraints.

7.2 Future Work

While this dissertation established the theoretical foundations and demonstrated feasibility on representative case studies, several research directions remain open to broaden applicability and enable industrial-scale deployment.

- **Scalability and tool integration.** Current constructions can handle systems with hundreds of states, but industrial cyber-physical systems (smart manufacturing, traffic networks, building automation) often involve thousands of interacting components. Three complementary approaches can address state-space explosion: (i) compositional reasoning under assume-guarantee contracts to decompose large models into verified sub-systems; (ii) symbolic state-space representation using binary decision diagrams or SMT solvers; (iii) on-the-fly verification that constructs only reachable states relevant to the property being checked. Integration with existing discrete-event system toolchains (Supremica, IDES, NuSMV) would accelerate adoption.

- **Active enforcement and controller synthesis.** The current framework focuses on analysis (verifying opacity, detecting faults), but many applications require active enforcement: designing supervisors that guarantee security or reliability by construction. For instance, industrial controllers should prevent timing side-channels by shaping observable behaviors; battery management systems should adjust charging schedules to avoid fault-prone regimes. Two synthesis approaches appear promising: (i) extending Ramadge-Wonham supervisory control theory [11] to SOA/SOAF with timed constraints, where supervisors disable transitions to enforce opacity or bounded detection delay; (ii) parameter co-design treating δ, output partitions, and fault windows as optimization variables to balance performance against security/reliability. This would transform the framework from an analysis tool into a design aid for timing-aware controllers.

- **Robustness under uncertainty.** The current framework assumes perfect clock synchronization, exact timing measurements, and deterministic behavior. Real systems face clock drift, sensor noise, environmental variations, and stochastic faults. Extending decidability to uncertain environments requires modeling uncertainty

while preserving tractability: (i) parametric robustness with bounded timing intervals (e.g., dwell times in $[\delta_{\min}, \delta_{\max}]$) and observers robust to all parameter values; (ii) stochastic extensions overlaying MDP/CTMC semantics to replace worst-case guarantees with probabilistic bounds; (iii) sensor fusion integrating Kalman or particle filtering to refine noisy observations. These extensions would enable deployment in wireless sensor networks, automotive systems, and other realistic environments.

- **Data-driven modeling and distributed verification.** Manual construction of SOA and SOAF models requires deep domain knowledge and becomes prohibitively labor-intensive for complex systems. Moreover, many modern cyber-physical systems are inherently distributed (smart grids, multi-robot teams, IoT deployments). Two orthogonal research directions address these challenges: (i) learning SOA/SOAF models from system traces using active or passive timed automata learning [147], [148], adapted to handle dwell times and set-valued outputs with formal PAC-style generalization guarantees; (ii) distributed opacity verification and diagnosis where multiple observers with partial visibility maintain local evolution automata and exchange messages to reach consensus on security violations or fault detections [60]. Data-driven approaches would lower the barrier to applying formal methods in domains lacking precise analytical models, while distributed algorithms would enable scalable deployment in large-scale networked systems.

List of Publications

This dissertation is based on the following publications:

Conference Papers

[1] T. Liu, C. Seatzu, and A. Giua, "Verification of Current State Opacity using Switching Output Automata," in *Proc. 2023 9th Int. Conf. Control, Decision and Information Technologies (CoDIT)*, Rome, Italy, Jul. 2023, pp. 2665–2670.

[2] T. Liu, C. Seatzu, and A. Giua, "Timed Opacity Verification for Switching Output Automata," in *Proc. 17th IFAC Workshop on Discrete Event Systems (WODES 2024)*, Rio de Janeiro, Brasil, 2024, vol. 58, no. 1, pp. 24–29.

[3] T. Liu, C. Seatzu, F. Pascucci, G. Cavone, and A. Giua, "Security-by-Design of Smart Water Supply Systems: a Switching Output Automaton-based Approach," in *Proc. 2024 IEEE 20th Int. Conf. Automation Science and Engineering (CASE)*, Bari, Italy, Aug. 2024, pp. 1532–1539.

[4] T. Liu, C. Seatzu, and A. Giua, "Timed Fault Diagnosis in Switching Output Automata," in *Proc. 2025 11th Int. Conf. Control, Decision and Information Technologies (CoDIT)*, Split, Croatia, 2025. (accepted)

References

[1] R. Rajkumar, I. Lee, L. Sha, and J. A. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proceedings of the 47th Design Automation Conference (DAC)*, 2010, pp. 731–736.

[2] D. A. Wollman, M. A. Weiss, Y.-S. Li-Baboud, E. Griffor, and M. Burns, "Framework for cyber-physical systems: Volume 3, timing annex," National Institute of Standards and Technology, Gaithersburg, MD, USA, NIST Special Publication (NIST SP) 1500-203, Sep. 2017.

[3] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*, 2nd ed. The MIT Press, 2017.

[4] C.-Y. Chen, S. Mohan, R. Pellizzoni, R. B. Bobba, and N. Kiyavash, "A novel side-channel in real-time schedulers," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 90–102.

[5] C.-Y. Chen, D. Sanyal, and S. Mohan, "Indistinguishability prevents scheduler side channels in real-time systems," in *Proc. 2021 ACM SIGSAC Conf. on Computer and Communications Security (CCS '21)*, Virtual Event, Republic of Korea, Nov. 2021, pp. 666–684.

[6] J. P. Hespanha and A. S. Morse, "Stability of switched systems with average dwell-time," in *Proc. 38th IEEE Conf. Decision and Control (CDC)*, vol. 3, Phoenix, AZ, USA, Dec. 1999, pp. 2655–2660.

[7] K. J. Åström and B. Wittenmark, "Discrete-time systems," in *Computer-Controlled Systems: Theory and Design*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 1997, ch. 2.

[8] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 3rd. Springer, 2021.

[9] W. M. Wonham and K. Cai, *Supervisory Control of Discrete-Event Systems*. Springer, 2018.

[10] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis, "Diagnosability of discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 40, no. 9, pp. 1555–1575, 1995.

[11] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete-event processes," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[12] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[13] T. A. Henzinger, "The theory of hybrid automata," in *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 1996, pp. 278–292.

[14] F. Cassez, "The dark side of timed opacity," in *Information Security and Privacy (ISA 2009)*, ser. LNCS, vol. 5576, Springer, 2009, pp. 21–30.

[15] K. Zhang, "State-based opacity of labeled real-time automata," *Theoretical Computer Science*, vol. 987, p. 114 373, 2024.

[16] A. Saboori and C. N. Hadjicostis, "Verification of initial-state opacity in security applications of discrete event systems," *IEEE Transactions on Automatic Control*, vol. 52, no. 9, pp. 1701–1706, 2007.

[17] A. Saboori and C. N. Hadjicostis, "Notions of security and opacity in discrete event systems," *46th IEEE Conference on Decision and Control*, pp. 5056–5061, 2007.

[18] A. Saboori and C. N. Hadjicostis, "Verification of K-step opacity and analysis of its complexity," *IEEE Transactions on Automation Science and Engineering*, vol. 8, no. 3, pp. 549–559, 2011.

[19] A. Saboori and C. N. Hadjicostis, "Verification of infinite-step opacity and complexity considerations," *IEEE Transactions on Automatic Control*, vol. 57, no. 5, pp. 1265–1269, 2012.

[20] F. Lin, "Opacity of discrete event systems and its applications," *Automatica*, vol. 47, no. 3, pp. 496–503, 2011.

[21] J. W. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan, "Opacity generalised to transition systems," *International Journal of Information Security*, vol. 7, no. 6, pp. 421–435, 2008.

[22] Y. Wu and S. Lafortune, "Comparative analysis of related notions of opacity in centralized and coordinated architectures," *Discrete Event Dynamic Systems*, vol. 23, no. 3, pp. 307–339, 2013.

[23] J. Balun and T. Masopust, "Comparing the notions of opacity for discrete-event systems," *Discrete Event Dynamic Systems*, vol. 31, no. 4, pp. 553–582, 2021.

[24] Y. Tong, Z. Li, C. Seatzu, and A. Giua, "Verification of language-based opacity in Petri nets using verifier," in *Proceedings of the American Control Conference (ACC)*, 2016, pp. 757–763.

[25] I. Saadaoui, Z. Li, N. Wu, and M. Khalgui, "Language-based opacity verification in partially observed Petri nets," *Mathematics*, vol. 11, no. 18, p. 3880, 2023.

[26] X. Cong, M. P. Fanti, A. M. Mangini, and Z. Li, "On-line verification of current-state opacity by Petri nets and integer linear programming," *ISA Transactions*, vol. 93, pp. 108–114, 2019.

[27] I. Saadaoui, Z. Li, and N. Wu, "Current-state opacity modelling and verification in partially observed Petri nets," *Automatica*, vol. 116, p. 108 907, 2020.

[28] A. Labed, M. Abid, M. A. A. El-Meligy, A. M. El-Sherbeeny, and Z. Li, "Current-state opacity verification in discrete event systems using labeled Petri nets," *Scientific Reports*, vol. 12, p. 21 645, 2022.

[29] Y. Tong, Z. Li, C. Seatzu, and A. Giua, "Verification of state-based opacity using Petri nets," *IEEE Transactions on Automatic Control*, vol. 62, no. 6, pp. 2823–2837, 2017.

[30] X. Yin and S. Lafortune, "A new approach for the verification of infinite-step and K-step opacity using two-way observers," *Automatica*, vol. 80, pp. 162–171, 2017.

[31] Z. Ma, Y. Tong, Z. Li, C. Seatzu, and A. Giua, "Strong infinite-step and k-step opacity using state recognizers," *Automatica*, vol. 131, p. 109 358, 2021.

[32] J. Balun and T. Masopust, "Verifying weak and strong k-step opacity in discrete-event systems," *Automatica*, vol. 155, p. 111 153, 2023.

[33] F. Cassez, J. Dubreil, and H. Marchand, "Synthesis of opaque systems with static and dynamic masks," *Formal Methods in System Design*, vol. 40, no. 1, pp. 88–115, 2012.

[34] T. Masopust and X. Yin, "Complexity of detectability, opacity and A-diagnosability for modular discrete-event systems," *Automatica*, vol. 101, pp. 290–295, 2019.

[35] J. Dubreil, P. Darondeau, and H. Marchand, "Opacity enforcing control synthesis," in *9th International Workshop on Discrete Event Systems, WODES 2008*, IEEE, 2008, pp. 28–35.

[36] X. Yin and S. Lafortune, "A new approach for synthesizing opacity-enforcing supervisors for partially-observed discrete-event systems," in *Proceedings of the American Control Conference (ACC)*, 2015.

[37] X. Yin and S. Lafortune, "Synthesis of maximally permissive supervisors for partially-observed discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 61, no. 5, pp. 1239–1254, 2016.

[38] X. Yin and S. Li, "Synthesis of dynamic masks for infinite-step opacity," *IEEE Transactions on Automatic Control*, vol. 65, no. 4, pp. 1429–1441, 2020.

[39] Y. Wu and S. Lafortune, "Synthesis of insertion functions for enforcement of opacity security properties," *Automatica*, vol. 50, no. 5, pp. 1336–1348, 2014.

[40] Y. Ji, Y. Wu, and S. Lafortune, "Enforcement of opacity by public and private insertion functions," *Automatica*, vol. 93, pp. 369–378, 2018.

[41] Y. Ji, X. Yin, and S. Lafortune, "Enforcing opacity by insertion functions under multiple energy constraints," *Automatica*, vol. 107, pp. 479–490, 2019.

[42] S. Lafortune, F. Lin, and C. N. Hadjicostis, "On the history of diagnosability and opacity in discrete event systems," *Annual Reviews in Control*, vol. 45, pp. 257–266, 2018.

[43] R. Jacob, J. Lesage, and J. Faure, "Overview of discrete event systems opacity: Models, validation, and quantification," *Annual Reviews in Control*, vol. 41, pp. 135–146, 2016.

[44] A. Wintenberg, M. Blischke, S. Lafortune, and N. Ozay, "A general language-based framework for specifying and verifying notions of opacity," *Discrete Event Dynamic Systems*, vol. 32, no. 2, pp. 253–289, 2022.

[45] L. Wang, N. Zhan, and J. An, "The opacity of real-time automata," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2845–2856, 2018.

[46] K. Zhang, "State-based opacity of real-time automata," in *27th International Conference on Implementation and Application of Automata (CIAA 2021)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 90, Schloss Dagstuhl, 2021, 12:1–12:13.

[47] J. Li, D. Lefebvre, C. N. Hadjicostis, and Z. Li, "Observers for a class of timed automata based on elapsed time graphs," *IEEE Transactions on Automatic Control*, vol. 67, no. 2, pp. 767–779, 2022.

[48] J. Li, D. Lefebvre, C. N. Hadjicostis, and Z. Li, "Verification of state-based timed opacity for constant-time labeled automata," *IEEE Transactions on Automatic Control*, vol. 70, no. 1, pp. 503–509, 2025.

[49] J. An, Q. Gao, L. Wang, N. Zhan, and I. Hasuo, "The opacity of timed automata," in *Proceedings of the 26th International Symposium on Formal Methods (FM 2024)*, ser. Lecture Notes in Computer Science, vol. 14933, Springer, 2024, pp. 620–637.

[50] J. Klein, P. Kogel, and S. Glesner, "Verifying opacity of discrete-timed automata," in *Proceedings of the IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE 2024)*, IEEE, 2024, pp. 55–65.

[51] É. André, D. Lime, D. Marinho, and J. Sun, "Guaranteeing timed opacity using parametric timed model checking," in *Formal Techniques for Distributed Objects, Components, and Systems - 42nd IFIP WG 6.1 International Conference, FORTE 2022*, ser. Lecture Notes in Computer Science, vol. 13273, Springer, 2022, pp. 115–135.

[52] I. Ammar, Y. E. Touati, M. Yeddes, and J. Mullins, "Bounded opacity for timed systems," *Journal of Information Security and Applications*, vol. 61, p. 102 926, 2021.

[53] É. André, S. Dépernet, and E. Lefaucheux, *The bright side of timed opacity*, 2024.

[54] S. Jiang, Z. Huang, V. Chandra, and R. Kumar, "A polynomial algorithm for testing diagnosability of discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 46, no. 8, pp. 1318–1321, 2001.

[55] T.-S. Yoo and S. Lafortune, "Polynomial-time verification of diagnosability of partially observed discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 47, no. 9, pp. 1491–1495, 2002.

[56] S. H. Zad, R. H. Kwong, and W. M. Wonham, "Fault diagnosis in discrete-event systems: Framework and model reduction," *IEEE Transactions on Automatic Control*, vol. 48, no. 7, pp. 1199–1212, 2003.

[57] F. Cassez and S. Tripakis, "The complexity of codiagnosability for discrete event and timed systems," *IEEE Transactions on Automatic Control*, vol. 57, no. 12, pp. 3158–3163, 2012.

[58] M. Sampath, S. Lafortune, and D. Teneketzis, "Active diagnosis of discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 43, no. 7, pp. 908–929, 1998.

[59] A. Paoli and S. Lafortune, "Safe diagnosability for fault-tolerant supervision of discrete-event systems," *Automatica*, vol. 41, no. 8, pp. 1335–1347, 2005.

[60] R. Debouk, S. Lafortune, and D. Teneketzis, "Coordinated decentralized protocols for failure diagnosis of discrete-event systems," *Discrete Event Dynamic Systems*, vol. 10, no. 1-2, pp. 33–86, 2000.

[61] W. Qiu and R. Kumar, "Decentralized failure diagnosis of discrete event systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, vol. 36, no. 2, pp. 384–395, 2006.

[62] Y. Wang, T.-S. Yoo, and S. Lafortune, "Diagnosis of discrete event systems using decentralized architectures," *Discrete Event Dynamic Systems*, vol. 17, no. 2, pp. 233–263, 2007.

[63] J. C. Basilio and S. Lafortune, "Robust codiagnosability of discrete event systems," in *Proceedings of the American Control Conference (ACC)*, 2009, pp. 2202–2209.

[64] F. Basile, P. Chiacchio, and G. D. Tommasi, "An efficient approach for online diagnosis of discrete event systems," *IEEE Transactions on Automatic Control*, vol. 54, no. 4, pp. 748–759, 2009.

[65] M. Dotoli, M. P. Fanti, A. M. Mangini, and W. Ukovich, "On-line fault detection in discrete event systems by petri nets and integer linear programming," *Automatica*, vol. 45, no. 11, pp. 2665–2672, 2009.

[66] M. P. Cabasino, A. Giua, S. Lafortune, and C. Seatzu, "A new approach for diagnosability analysis of petri nets using verifier nets," *IEEE Transactions on Automatic Control*, vol. 57, no. 12, pp. 3104–3117, 2012.

[67] A. Ram'irez-Trevi no, E. Ruiz-Beltr'an, I. Rivera-Rangel, and E. L'opez-Mellado, "Online fault diagnosis of discrete event systems: A petri net-based approach," *IEEE Transactions on Automation Science and Engineering*, vol. 4, no. 1, pp. 31–39, 2007.

[68] G. Jiroveanu and R. K. Boel, "The diagnosability of petri net models using minimal explanations," *IEEE Transactions on Automatic Control*, vol. 55, no. 7, pp. 1663–1668, 2010.

[69] A. Benveniste, E. Fabre, S. Haar, and C. Jard, "Diagnosis of asynchronous discrete-event systems: A net unfolding approach," *IEEE Transactions on Automatic Control*, vol. 48, no. 5, pp. 714–727, 2003.

[70] L. K. Carvalho, J. C. Basilio, and M. V. Moreira, "Robust diagnosis of discrete-event systems against intermittent loss of observations," *Automatica*, vol. 48, no. 9, pp. 2068–2078, 2012.

[71] L. K. Carvalho, M. V. Moreira, and J. C. Basilio, "Robust diagnosis of discrete-event systems against permanent loss of observations," *Automatica*, vol. 49, no. 1, pp. 223–231, 2013.

[72] J. H. A. Tomola, F. G. Cabral, L. K. Carvalho, and J. C. Basilio, "Robust disjunctive-codiagnosability of discrete-event systems against permanent loss of observations," *IEEE Transactions on Automatic Control*, vol. 62, no. 11, pp. 5808–5815, 2017.

[73] S. Takai, "A general framework for diagnosis of discrete event systems subject to sensor failures," *Automatica*, vol. 129, p. 109 669, 2021.

[74] A. Wada and S. Takai, "Decentralized diagnosis of discrete event systems subject to permanent sensor failures," *Discrete Event Dynamic Systems*, vol. 32, pp. 159–193, 2022.

[75] J. Zaytoon and S. Lafortune, "Overview of fault diagnosis methods for discrete event systems," *Annual Reviews in Control*, vol. 37, no. 2, pp. 308–320, 2013.

[76] S. Tripakis, "Fault diagnosis for timed automata," in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer, 2002, pp. 205–224.

[77] P. Bouyer, F. Chevalier, and D. D'Souza, "Fault diagnosis using timed automata," *Lecture Notes in Computer Science*, vol. 3441, pp. 219–233, 2005.

[78] M. Ghazel, A. Toguyéni, and P. Yim, "Fault diagnosis graph for time petri nets," *European Journal of Control*, vol. 15, no. 3-4, pp. 324–338, 2009.

[79] S. Hashtrudi Zad, R. H. Kwong, and W. M. Wonham, "Fault diagnosis in discrete-event systems: Incorporating timing information," *IEEE Transactions on Automatic Control*, vol. 50, no. 7, pp. 1010–1015, 2005.

[80] V. Puig, C. Ocampo-Martínez, R. Pérez, G. Cembrano, J. Quevedo, and T. Escobet, "Fault diagnosis using a timed discrete-event approach based on interval observers: Application to sewer networks," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 40, no. 5, pp. 900–916, 2010.

[81] Y. Pencolé and A. Subias, "A chronicle-based diagnosability approach for discrete timed-event systems: Application to web services," in *Proceedings of the 20th International Workshop on Principles of Diagnosis (DX)*, 2009, pp. 133–140.

[82] European Parliament and Council, *Recital 35 - health data*, General Data Protection Regulation (GDPR), 2016.

[83] F. K. Dankar and K. El Emam, "Addressing contemporary threats in anonymised healthcare data using privacy engineering," *npj Digital Medicine*, vol. 4, no. 1, p. 37, 2021.

[84] S. A. Haque, S. Aziz, and M. Rahman, "Medical cyber-physical systems: A survey," *Journal of Medical Systems*, vol. 42, no. 4, p. 74, 2018.

[85] R. Taormina, S. Galelli, N. O. Tippenhauer, E. Salomons, and A. Ostfeld, "Characterizing cyber-physical attacks on water distribution systems," *Journal of Water Resources Planning and Management*, vol. 143, no. 5, p. 04 017 009, 2017.

[86] G. Loukas, *Cyber-physical attacks: A growing invisible threat*. Butterworth-Heinemann, 2015.

[87] P. Chalupa, J. Novák, and V. Bobál, "Comprehensive model of dts200 three tank system in simulink," *International Journal of Mathematical Models and Methods in Applied Sciences*, vol. 6, no. 2, pp. 358–365, 2012.

[88] S. X. Ding, *Model-based Fault Diagnosis Techniques: Design Schemes, Algorithms, and Tools*. Springer Berlin, Heidelberg, 2008.

[89] B. J. Drew, P. Harris, J. K. Zègre-Hemsey, *et al.*, "Insights into the problem of alarm fatigue with physiologic monitor devices: A comprehensive observational study of consecutive intensive care unit patients," *PLOS ONE*, vol. 9, no. 10, e110274, 2014.

[90] A. E. W. Johnson, T. J. Pollard, L. Shen, *et al.*, "MIMIC-III, a freely accessible critical care database," *Scientific Data*, vol. 3, p. 160 035, 2016.

[91] Y. Sun, F. P.-W. Lo, and B. Lo, "Security and privacy for the internet of medical things enabled healthcare systems: A survey," *IEEE Access*, vol. 7, pp. 183 339–183 355, 2019.

[92] P. A. H. Williams and A. J. Woodward, "Cybersecurity vulnerabilities in medical devices: A complex environment and multifaceted problem," *Medical Devices (Auckland, N.Z.)*, vol. 8, pp. 305–316, 2015.

[93] R. C. Barrows Jr. and P. D. Clayton, "Privacy, confidentiality, and electronic medical records," *Journal of the American Medical Informatics Association*, vol. 3, no. 2, pp. 139–148, 1996.

[94] T. H. McCoy and R. H. Perlis, "Temporal trends and characteristics of reportable health data breaches, 2010-2017," *JAMA*, vol. 320, no. 12, pp. 1282–1284, 2018.

[95] G. Bai, J. X. Jiang, and R. Flasher, "Hospital risk of data breaches," *JAMA Internal Medicine*, vol. 177, no. 6, pp. 878–880, 2017.

[96] A. A. Boxwala, J. Kim, J. M. Grillo, and L. Ohno-Machado, "Using statistical and machine learning to help institutions detect suspicious access to electronic health records," *Journal of the American Medical Informatics Association*, vol. 18, no. 4, pp. 498–505, 2011.

[97] J. Li, K.-Y. Huang, J. Jin, and J. Shi, "A survey on statistical methods for health care fraud detection," *Health Care Management Science*, vol. 11, no. 3, pp. 275–287, 2008.

[98] H. T. Neprash, C. C. McGlave, D. A. Cross, B. A. Virnig, H. A. Huskamp, and P. Karaca-Mandic, "Trends in ransomware attacks on US hospitals, clinics, and other health care delivery organizations, 2016-2021," *JAMA Health Forum*, vol. 3, no. 12, e224873, 2022.

[99] K. El Emam, E. Jonker, L. Arbuckle, and B. Malin, "A systematic review of re-identification attacks on health data," *PLOS ONE*, vol. 6, no. 12, e28071, 2011.

[100] A. Gkoulalas-Divanis, G. Loukides, and J. Sun, "Publishing data from electronic health records while preserving privacy: A survey of algorithms," *Journal of Biomedical Informatics*, vol. 50, pp. 4–19, 2014.

[101] G. J. Annas, "Medical privacy and medical research–judging the new federal regulations," *New England Journal of Medicine*, vol. 346, no. 3, pp. 216–220, 2002.

[102] B. D. Winters, M. M. Cvach, C. P. Bonafide, *et al.*, "Technological distractions (part 2): A summary of approaches to manage clinical alarms with intent to reduce alarm fatigue," *Critical Care Medicine*, vol. 46, no. 1, pp. 130–137, 2018.

[103] A.-S. Poncette, M. M. Wunderlich, J. J. Grunow, *et al.*, "Patient monitoring alarms in an intensive care unit: Observational study with do-it-yourself instructions," *Journal of Medical Internet Research*, vol. 23, no. 5, e26494, May 2021.

[104] S. Sendelbach and M. Funk, "Alarm fatigue: A patient safety concern," *AACN Advanced Critical Care*, vol. 24, no. 4, pp. 378–386, 2013.

[105] M. Cvach, "Monitor alarm fatigue: An integrative review," *Biomedical Instrumentation & Technology*, vol. 46, no. 4, pp. 268–277, 2012.

[106] C. W. Paine, V. V. Goel, E. Ely, *et al.*, "Systematic review of physiologic monitor alarm characteristics and pragmatic interventions to reduce alarm frequency," *Journal of Hospital Medicine*, vol. 11, no. 2, pp. 136–144, 2016.

[107] M. M. Gunal and M. Pidd, "Discrete event simulation for performance modelling in health care: A review of the literature," *Journal of Simulation*, vol. 4, no. 1, pp. 42–51, 2010.

[108] J. Karnon, J. Stahl, A. Brennan, J. J. Caro, J. Mar, and J. Möller, "Modeling using discrete event simulation: A report of the ISPOR-SMDM modeling good research practices task force—4," *Value in Health*, vol. 15, no. 6, pp. 821–827, 2012.

[109] U. Siebert, O. Alagoz, A. M. Bayoumi, *et al.*, "State-transition modeling: A report of the ISPOR-SMDM modeling good research practices task force—3," *Value in Health*, vol. 15, no. 6, pp. 812–820, 2012.

[110] X. Yin and S. Lafortune, "A new approach for the verification of infinite-step and K-step opacity using two-way observers," *Automatica*, vol. 80, pp. 162–171, 2017.

[111] K. Kawamoto, C. A. Houlihan, E. A. Balas, and D. F. Lobach, "Improving clinical practice using clinical decision support systems: A systematic review of trials to identify features critical to success," *BMJ*, vol. 330, no. 7494, pp. 765–768, 2005.

[112] A. X. Garg, N. K. J. Adhikari, H. McDonald, *et al.*, "Effects of computerized clinical decision support systems on practitioner performance and patient outcomes: A systematic review," *JAMA*, vol. 293, no. 10, pp. 1223–1238, 2005.

[113] A. Rajkomar, E. Oren, K. Chen, *et al.*, "Scalable and accurate deep learning with electronic health records," *npj Digital Medicine*, vol. 1, p. 18, 2018.

[114] R. Miotto, L. Li, B. A. Kidd, and J. T. Dudley, "Deep patient: An unsupervised representation to predict the future of patients from the electronic health records," *Scientific Reports*, vol. 6, p. 26 094, 2016.

[115] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini, "Security, privacy and trust in Internet of Things: The road ahead," *Computer Networks*, vol. 76, pp. 146–164, 2015.

[116] L. Lu, X. Han, J. Li, J. Hua, and M. Ouyang, "A review on the key issues for lithium-ion battery management in electric vehicles," *Journal of Power Sources*, vol. 226, pp. 272–288, 2013.

[117] Q. Wang, P. Ping, X. Zhao, G. Chu, J. Sun, and C. Chen, "Thermal runaway caused fire and explosion of lithium ion battery," *Journal of Power Sources*, vol. 208, pp. 210–224, 2012.

[118] T. M. Bandhauer, S. Garimella, and T. F. Fuller, "A critical review of thermal issues in lithium-ion batteries," *Journal of The Electrochemical Society*, vol. 158, no. 3, R1–R25, 2011.

[119] X. Feng, M. Ouyang, X. Liu, L. Lu, Y. Xia, and X. He, "Thermal runaway mechanism of lithium ion battery for electric vehicles: A review," *Energy Storage Materials*, vol. 10, pp. 246–267, 2018.

[120] S. E. Mahdy *et al.*, "Quantitative evaluation of thermal runaway in lithium-ion batteries under critical heating conditions to enhance safety," *Scientific Reports*, 2025.

[121] Z. Liao, S. Zhang, K. Li, G. Zhang, and T. G. Habetler, "A survey of methods for monitoring and detecting thermal runaway of lithium-ion batteries," *Journal of Power Sources*, vol. 436, p. 226 879, 2019.

[122] D. Kong, H. Lv, P. Ping, and G. Wang, "A review of early warning methods of thermal runaway of lithium ion batteries," *Journal of Energy Storage*, vol. 64, p. 107 073, 2023.

[123] G. L. Plett, "Sigma-point Kalman filtering for battery management systems of LiPB-based HEV battery packs: Part 1: Introduction and state estimation," *Journal of Power Sources*, vol. 161, no. 2, pp. 1356–1368, 2006.

[124] X. Hu, K. Zhang, K. Liu, X. Lin, S. Dey, and S. Onori, "Advanced fault diagnosis for lithium-ion battery systems: A review of fault mechanisms, fault features, and diagnosis procedures," *IEEE Industrial Electronics Magazine*, 2020.

[125] R. Xiong, S. Ma, H. Li, F. Sun, and J. Li, "Toward a safer battery management system: A critical review on diagnosis and prognosis of battery short circuit," *iScience*, vol. 23, no. 4, p. 101 010, 2020.

[126] G. L. Plett, "Extended Kalman filtering for battery management systems of LiPB-based HEV battery packs: Part 1. background," *Journal of Power Sources*, vol. 134, no. 2, pp. 252–261, 2004.

[127] R. Xiong, W. Sun, Q. Yu, and F. Sun, "Research progress, challenges and prospects of fault diagnosis on battery system of electric vehicles," *Applied Energy*, vol. 279, p. 115 855, 2020.

[128] S. Dey, H. E. Perez, and S. J. Moura, "Model-based battery thermal fault diagnostics: Algorithms, analysis, and experiments," *IEEE Transactions on Control Systems Technology*, vol. 27, no. 2, pp. 576–587, 2019.

[129] C. Zheng, Z. Chen, T. Lin, D. Huang, and S. Zhou, "Sensor fault diagnosis method for lithium-ion battery packs based on stochastic hybrid automata and unscented particle filter," *Energy*, vol. 191, p. 116 504, 2020.

[130] T. Lin, Z. Chen, C. Zheng, D. Huang, and S. Zhou, "Fault diagnosis of lithium-ion battery pack based on hybrid system and dual extended Kalman filter algorithm," *IEEE Transactions on Transportation Electrification*, vol. 7, no. 1, pp. 26–36, 2021.

[131] Z. Chen, C. Zheng, T. Lin, and Q. Yang, "Multifault diagnosis of Li-ion battery pack based on hybrid system," *IEEE Transactions on Transportation Electrification*, vol. 8, no. 2, pp. 1769–1784, 2022.

[132] X. Lu, M. Zhou, A. C. Ammari, and J. Ji, "Hybrid Petri nets for modeling and analysis of microgrid systems," *IEEE/CAA Journal of Automatica Sinica*, vol. 3, no. 4, pp. 349–356, 2016.

[133] Z. Jiang, Z. Li, N. Wu, and M. Zhou, "A Petri net approach to fault diagnosis and restoration for power transmission systems to avoid the output interruption of substations," *IEEE Systems Journal*, vol. 12, no. 3, pp. 2566–2576, 2018.

[134] K. Zhang, X. Hu, Y. Liu, X. Lin, and W. Liu, "Multi-fault detection and isolation for lithium-ion battery systems," *IEEE Transactions on Power Electronics*, vol. 37, no. 1, pp. 971–989, 2022.

[135] Z. Chen, F. Lin, C. Wang, Y. L. Wang, and M. Xu, "Active diagnosability of discrete event systems and its application to battery fault diagnosis," *IEEE Transactions on Control Systems Technology*, vol. 22, no. 5, pp. 1892–1898, 2014.

[136] J. Jaguemont and F. Bardé, "A critical review of lithium-ion battery safety testing and standards," *Applied Thermal Engineering*, vol. 231, p. 121 014, 2023.

[137] X. Lai, C. Jin, W. Yi, *et al.*, "Mechanism, modeling, detection, and prevention of the internal short circuit in lithium-ion batteries: Recent advances and perspectives," *Energy Storage Materials*, 2021.

[138] Y. Wu *et al.*, "Improving battery safety by early detection of internal shorting with a bifunctional separator," *Nature Communications*, vol. 5, p. 5193, 2014.

[139] W. Mei *et al.*, "Operando monitoring of thermal runaway in commercial lithium-ion cells via advanced lab-on-fiber technologies," *Nature Communications*, vol. 14, no. 1, p. 5251, 2023.

[140] C. Forgez, D. Vinh Do, G. Friedrich, M. Morcrette, and C. Delacourt, "Thermal modeling of a cylindrical LiFePO4/graphite lithium-ion battery," *Journal of Power Sources*, vol. 195, no. 9, pp. 2961–2968, 2010.

[141] X. Feng *et al.*, "Thermal runaway features of large format prismatic lithium ion battery using extended volume accelerating rate calorimetry," *Journal of Power Sources*, vol. 255, pp. 294–301, 2014.

[142] D. Ren *et al.*, "A comparative investigation of aging effects on thermal runaway behavior of lithium-ion batteries," *eTransportation*, vol. 2, p. 100 034, 2019.

[143] T. Waldmann, M. Wilka, M. Kasper, M. Fleischhammer, and M. Wohlfahrt-Mehrens, "Temperature dependent ageing mechanisms in lithium-ion batteries – a post-mortem study," *Journal of Power Sources*, vol. 262, pp. 129–135, 2014.

[144] J. Vetter, P. Novák, M. R. Wagner, *et al.*, "Ageing mechanisms in lithium-ion batteries," *Journal of Power Sources*, vol. 147, pp. 269–281, 2005.

[145] M. Petzl, M. Kasper, and M. A. Danzer, "Lithium plating in a commercial lithium-ion battery – a low-temperature aging study," *Journal of Power Sources*, vol. 275, pp. 799–807, 2015.

[146] G. L. Plett, *Battery Management Systems, Volume I: Battery Modeling*. Boston: Artech House, 2015.

[147] F. Vaandrager, "Model learning," *Communications of the ACM*, vol. 60, no. 2, pp. 86–95, 2017.

[148] M. Tappler, B. K. Aichernig, K. G. Larsen, and F. Lorber, "Time to learn – learning timed automata from tests," in *Formal Modeling and Analysis of Timed Systems (FORMATS)*, Springer, 2019, pp. 216–235.