Incremental Security Enforcement for Underwater Drones

Thesis submitted to the
Indian Institute of Technology Bhubaneswar
for partial fulfillment of the requirements
for the degree of

of

Bachelor and Master of Technology
in
Computer Science and Engineering

by

Vedanta Mohapatra

Under the guidance of

Dr. Srinivas Pinisetty

DEPARTMENT OF COMPUTER SC. & ENGINEERING
SCHOOL OF ELECTRICAL AND COMPUTER SCIENCES
INDIAN INSTITUTE OF TECHNOLOGY BHUBANESWAR
November 2024

APPROVAL OF THE VIVA-VOCE BOARD

DD/MM/YYYY

Certified that the thesis entitled **Incremental Security Enforcement for Underwater Drones**, submitted by **Vedanta Mohapatra** to the Indian Institute of Technology Bhubaneswar, for the award of the degree of Doctor of Philosophy has been accepted by the external examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

(Member of DSC) (Member of DSC) (Member of DSC)

(Supervisor)

(External Examiner) (Chairman)

CERTIFICATE

This is to certify that the thesis entitled **Write Thesis Title Here**, submitted by **Student Name** to Indian Institute of Technology Bhubaneswar, is a record of bonafide research work under my supervision and I consider it worthy of consideration for the award of the degree of Doctor of Philosophy of the Institute.

Date :

Supervisor Name
Designation
School of ******* Sciences
Indian Institute of Technology Bhubaneswar
Bhubaneswar, India

DECLARATION

I certify that

a. the work contained in the thesis is original and has been done by myself under the general supervision of my supervisor.

b. the work has not been submitted to any other institute for any degree or diploma.

c. I have followed the guidelines provided by the institute in writing the thesis.

d. I have conformed to the norms and guidelines given in the ethical code of conduct of the institute.

e. whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

f. whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Student Name

Acknowledgments

Write Acknowledgments Here

Student Name

Abstract

Cyber-Physical Systems (CPS) are integral to a wide range of critical applications, including autonomous vehicles, medical devices, and industrial automation. These systems merge physical processes with computational elements, creating complex environments that require stringent safety, security, and performance guarantees. However, the dynamic nature of CPS, combined with unpredictable environmental interactions, introduces significant challenges in ensuring system correctness at runtime.

Runtime enforcement plays a crucial role in addressing these challenges by monitoring and dynamically adjusting the system's behavior to adhere to predefined safety and security policies. Unlike static verification techniques, which may not account for unforeseen scenarios or evolving threats, runtime enforcement provides a real-time safeguard that detects and mitigates violations as they occur. This proactive approach enhances the resilience of CPS, ensuring they remain robust against internal faults and external attacks, ultimately protecting both the system's operation and its users.

In this work, we explore the use of compositional runtime enforcement in underwater robotic swarms, comparing its effectiveness against the traditional monolithic approach. The study focuses on applying runtime enforcement to a Multi-Robot Coverage Path Planning (MRCPP) algorithm, specifically designed for mapping underwater vegetation within a seagrass bed. By examining both approaches, we aim to assess their impact on the algorithm's performance, scalability, and adaptability in dynamic underwater environments.

Contents

List of Figures

List of Tables

List of Abbreviations

5G	Fifth generation
ABER	Average bit error rate
ADSL	Asymmetric digital subscriber line
AF	Amplify-and-forward
ASER	Average symbol error rate
AWGN	Additive white Gaussian noise
BER	Bit error rate
BFSK	Binary frequency shift keying
BPSK	Binary phase shift keying

List of Symbols

$\lvert \cdot \rvert$	Absolute value
$\begin{pmatrix} k \\ l \end{pmatrix}$	Binomial coefficient
$B(\cdot, \cdot)$	Beta function
$\Phi_2^{(n)}(\cdot)$	Confluent form of the generalized Lauricella series
${}_1F_1(\cdot, \cdot; \cdot)$	Confluent hypergeometric function
$F_X(\cdot)$	Cumulative distribution function of random variable X
$\mathbb{E}[\cdot]$	Expectation operator
$\exp(\cdot)$	Exponential

Chapter 1

Introduction

Cyber-physical systems (CPSs) in the context of Underwater Unmanned Vehicles (UUVs) incorporate distributed embedded controllers that manage various physical processes critical to underwater operations [1]. These systems have become an integral part of modern maritime applications, such as underwater exploration, environmental monitoring, military reconnaissance, and offshore infrastructure inspection. The security of CPSs controlling UUVs has emerged as a significant concern due to the critical nature of their applications. As these systems grow increasingly complex and interconnected, they become more susceptible to sophisticated cyberattacks [2, 3].

In cyber-physical security attacks targeting UUVs, remote attackers can gain unauthorized control over the vehicle, interfere with its physical processes, and potentially cause catastrophic damage, including the loss of sensitive data, destruction of assets, or even risks to human life. For instance, malicious attacks on UUVs used in defense operations could lead to compromised missions or disruption of naval fleets. Recently, there were massive cyber attacks on Iran's nuclear facilities and government agencies [4]. Literature highlights notable CPS attacks, such as the Stuxnet worm damaging Iranian centrifuges [5], the Maroochy Shire Water Services attack [6], and the German Steel Mill attack [7]. Similarly, cyberattacks on autonomous drones and underwater vehicles have been documented, emphasizing the growing threat landscape for UUV

operations [8].

To address these challenges, formal runtime enforcement techniques [9–12] have been proposed as reliable mechanisms for mitigating security concerns in CPSs, including UUVs [13–15]. The research domain known as runtime verification (RV) [16] focuses on dynamically verifying a set of desirable policies during the execution of a "black-box" system. An RV monitor observes the system's execution trace without interfering and determines whether it satisfies or violates specific policies.

Runtime enforcement (RE), an active counterpart to passive runtime verification, provides a mechanism to guarantee the adherence to desired policies during system execution. In RE mechanisms, an enforcer is designed to monitor a black-box system and take corrective actions when policy violations are detected. For UUVs, such actions are critical for ensuring mission safety and security. Evasive measures include blocking harmful actions [9], modifying input sequences by suppressing or adding actions [11], or buffering inputs until they can be safely executed [10, 12]. These techniques are particularly relevant for UUVs operating in dynamic and unpredictable underwater environments, where maintaining system safety and security is paramount.

1.1 Literature Survey

Runtime verifiers are extensively used in underwater drone systems to ensure operational accuracy and detect malicious attacks. For example, similar to the Argus framework [17], external intelligent controllers can monitor the physical processes of underwater drones, enforcing policies such as depth, velocity, and collision avoidance based on operational invariants. Alternatively, runtime verification can be embedded directly within controllers [18], enabling real-time detection and reporting of anomalies or rule violations.

Specific cyberattacks on underwater drones include jamming, injection, and data alteration. To counter such threats, various runtime enforcement (RE) mechanisms

have been proposed. For instance, Baird et al. [19] modeled these attacks in a simulated drone system and developed runtime enforcers. Additionally, resilient control methods, such as those by Sun et al. [20], use dual-mode algorithms to counter denial-of-service (DoS) attacks in CPS environments.

Traditional RE approaches (e.g., [10,11]) rely on buffering or editing event sequences but are less suitable for reactive systems like underwater drones, where actions must occur instantaneously. Recent frameworks such as shields [21,22] allow for bi-directional enforcement by transforming outputs dynamically while maintaining system reactivity.

Our research builds on these principles, proposing a compositional RE framework for underwater drones. This approach allows enforcers to be incrementally added in series, enabling seamless integration of new security policies as threats evolve, without disrupting the existing system. This compositionality ensures flexibility and adaptability in securing underwater drones against emerging challenges.

1.2 Motivation

In practice, runtime security policies for underwater drones evolve over time as cyber-attacks emerge and software systems grow more complex. Understanding the secured and unsecured behaviors of such systems also evolves with advancements in their applications. Underwater drones, deployed in diverse areas such as ocean exploration, environmental monitoring, and military surveillance, often require new and more sophisticated security policies alongside existing ones. Ensuring the security of these systems is inherently incremental: as new threats are identified, new security measures and patches are implemented to address them.

Consider an underwater drone swarm as an example of a cyber-physical system (CPS). These drones operate with distributed controllers that manage their physical processes, such as depth, orientation, and velocity. They are frequently employed to monitor marine ecosystems, detect underwater anomalies, and transport goods in

aquatic environments. Each drone operates within specific underwater zones and must adhere to restrictions in spatial movement to avoid collisions, boundary violations, or excessive energy consumption. However, the increasing use of underwater drones has also introduced new security vulnerabilities, exposing them to cyberattacks such as spoofing, jamming, and unauthorized access. Research has highlighted the necessity of augmenting existing security frameworks with more robust policies to protect against such threats.

Incremental enforcement of security policies becomes essential in this context, especially as new challenges or attack vectors emerge. Instead of synthesizing a monolithic enforcer from scratch for every new policy, incremental enforcement allows for the addition of new policies without altering or revalidating the existing ones. This approach is particularly relevant when the underlying system is designed to handle dynamic, evolving environments, as is often the case with underwater drones. Monolithic enforcement methods face significant drawbacks, including high costs of redevelopment, re-certification, and risks of disrupting previously concealed or secret policies. These issues underscore the importance of designing enforcers capable of incorporating new policies without requiring knowledge of previously enforced ones.

This work focuses on studying incremental enforcement mechanisms for underwater drones, utilizing approaches like those proposed in prior research (e.g., [22, 23]), which are suitable for reactive CPS systems. In this context, the underwater drones are modeled as synchronous reactive systems. These systems continuously interact with their environment, with execution consisting of steps where the system reads inputs, performs reactions, and computes outputs. Such modeling ensures that new security policies can be incrementally added and enforced without compromising the system's overall integrity or performance.

Chapter 2

Runtime Enforcement Framework

Before delving into the specific details of the runtime enforcement framework, it is important to first establish some foundational concepts and the necessary notation. This section introduces key elements such as edit functions, runtime enforcement for synchronous programs, and the framework for policies defined as safety automata. Understanding these preliminaries will provide the necessary context for the subsequent discussions on the design and application of runtime enforcement mechanisms in various program types.

2.1 Preliminaries and Notation

In this section, we introduce the notations and the safety automaton formalism used to define policies to be monitored and enforced. We also briefly recall the RE problem for synchronous programs (all the constraints that an enforcer should fulfill).

A finite word over a finite alphabet Σ is a finite sequence $\sigma = a_1 \cdot a_2 \cdots a_n$ of members of Σ, and Σ^* denotes the set of finite words over Σ. Considering a finite word σ, its length is denoted as $|\sigma|$. ϵ_Σ is used to denote the empty word over Σ is denoted by ϵ_Σ, or ϵ (when the context makes it evident). Given two words σ and σ', their *concatenation*

is indicated as $\sigma \cdot \sigma'$. A word σ' is a *prefix* of a word σ, represented as $\sigma' \preccurlyeq \sigma$, whenever a word σ'' is present such that $\sigma = \sigma' \cdot \sigma''$; σ is called an *extension* of σ'.

A reactive system with a finite ordered sets of Boolean inputs $I = \{i_1, i_2, \cdots, i_n\}$ and Boolean outputs $O = \{o_1, o_2, \cdots, o_m\}$ is considered. $\Sigma_I = 2^I$ denotes the input alphabet, $\Sigma_O = 2^O$ denotes the output alphabet, and the input-output alphabet is $\Sigma = \Sigma_I \times \Sigma_O$. A bit-vector/complete monomial will be used to represent each input (resp. output) event. For example, let us consider $I = \{P, Q\}$. Then, the input $\{P\} \in \Sigma_I$ is denoted as 10, while $\{Q\} \in \Sigma_I$ is denoted as 01 and $\{P, Q\} \in \Sigma_I$ is denoted as 11. A reaction (or input-output event) has the following structure: (x_i, y_i), where $x_i \in \Sigma_I$ and $y_i \in \Sigma_O$.

Given $\sigma = (x_1, y_1) \cdot (x_2, y_2) \cdots (x_n, y_n) \in \Sigma^*$ which is an input-output word, the input word acquired from σ is $\sigma_I = x_1 \cdot x_2 \cdots x_n \in \Sigma_I$, which is a projection that ignores outputs and is based on inputs. Similarly, the output word obtained from σ is $\sigma_O = y_1 \cdot y_2 \cdots y_n \in \Sigma_O$ is the projection on outputs ignoring inputs.

A policy denoted as φ (over Σ) represents a set $\mathcal{L}(\varphi) \subseteq \Sigma^*$. Given a word $\sigma \in \Sigma^*$, $\sigma \models \varphi$ iff $\sigma \in \mathcal{L}(\varphi)$. A policy φ is *prefix-closed* if all prefixes of all words from $\mathcal{L}(\varphi)$ are also in $\mathcal{L}(\varphi)$: $\mathcal{L}(\varphi) = \{w \mid \exists w' \in \mathcal{L}(\varphi) : w \preccurlyeq w'\}$. Prefix-closed policies are the focus of this study. Policies are formalized as safety automata, which we define next in this section.

Synchronous programming languages [24] are ideal for developing synchronous reactive systems. They express safety properties via observers [25], which are statically verified (using model checking). Safety automata are analogous to observers but are enforced at runtime.

Definition 1 (Safety Automaton)**.** *A* safety automaton *(SA)* $\mathcal{A} = (Q, q_0, q_v, \Sigma, \rightarrow)$ *is a tuple, where* Q *denotes the set of states, known as* locations*,* $q_0 \in Q$ *is a distinct starting location,* $q_v \in Q$ *is a distinct non-accepting (violating) location, the alphabet is* $\Sigma = \Sigma_I \times \Sigma_O$*, and the transition relation is* $\rightarrow \subseteq Q \times \Sigma \times Q$*. Except for* q_v*, all the other locations are accepting (i.e., all the locations in* $Q \setminus \{q_v\}$*). Location* q_v *is a distinct*

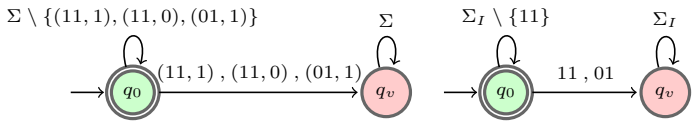
violating (trap) location, thus no transitions in $\rightarrow$ *from* q_v *to a location in* $Q \setminus \{q_v\}$. *Whenever there exists* $(q, a, q') \in \rightarrow$, *we denote it as* $q \xrightarrow{a} q'$. *Relation* $\rightarrow$ *is extended to words* $\sigma \in \Sigma^*$ *by noting* $q \xrightarrow{\sigma.a} q'$ *whenever there exists* q'' *such that* $q \xrightarrow{\sigma} q''$ *and* $q'' \xrightarrow{a} q'$. *A location* $q \in Q$ *is reachable from* q_0 *if there exists a word* $\sigma \in \Sigma^*$ *such that* $q_0 \xrightarrow{\sigma} q$.

An SA $\mathcal{A} = (Q, q_0, q_v, \Sigma, \rightarrow)$ is *deterministic* if $\forall q \in Q, \forall a \in \Sigma, (q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'') \implies (q' = q'')$. $\mathcal{A}$ is *complete* if $\forall q \in Q, \forall a \in \Sigma, \exists q' \in Q, q \xrightarrow{a} q'$. A word σ is *accepted* by $\mathcal{A}$ if there exists $q \in Q \setminus \{q_v\}$ such that $q_0 \xrightarrow{\sigma} q$. The set of all words accepted by $\mathcal{A}$ is denoted as $\mathcal{L}(\mathcal{A})$.

Remark 1. *We can first determinize and complete a non-deterministic or incomplete automaton provided by the user. We further assume that* Q *has no (redundant) locations that are unreachable from* q_0. *Hence, in the rest of this work,* φ *is a safety policy specified as deterministic and complete SA* $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$.

The enforcer must first alter inputs from the environment in each step according to policy φ specified as SA $\mathcal{A}_\varphi$ according to the causality requirement. As a result, we must examine the input policy obtained by projecting on inputs from $\mathcal{A}_\varphi$.

Definition 2 (Input SA $\mathcal{A}_{\varphi_I}$)**.** *Given* $\varphi \subseteq \Sigma^*$, *specified as SA* $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, *by discarding outputs on the transitions, input SA* $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ *is derived from* $\mathcal{A}_\varphi$. *That is, for every transition* $q \xrightarrow{(x,y)} q' \in \rightarrow$ *where* $(x, y) \in \Sigma$, *there is a transition* $q \xrightarrow{x} q' \in \rightarrow_I$, *where* $x \in \Sigma_I$. $\mathcal{L}(\mathcal{A}_{\varphi_I})$ *is represented as* $\varphi_I \subseteq \Sigma_I^*$.



(a) SA $\mathcal{A}_P$. (b) Input SA from $\mathcal{A}_P$.

Figure 2.1: SA (left), and its input SA (right). [1]

Example 1 (Example policy defined as SA and its input SA)**.** *Consider* $I = \{B, Q\}$ *and* $O = \{X\}$. *Let us consider the policy: P: "B and Q can't happen at the same*

[1] Here, $\Sigma = \{(00, 0), (00, 1), (01, 0), (01, 1), (10, 0), (10, 1), (11, 0), (11, 1)\}$. So $\Sigma \setminus \{(11, 1), (11, 0), (01, 1)\} = \{(00, 0), (00, 1), (01, 0), (10, 0), (10, 1)\}$.

time, and Q and X can't happen at the same time". Policy P is defined by the safety automaton in Figure 2.1a. The input SA for the SA in Figure 2.1a defining policy P is shown in Figure 2.1b. Though the SA $\mathcal{A}_\varphi$ is deterministic, the input SA $\mathcal{A}_{\varphi_I}$ may be non-deterministic. This is the case with the considered example as shown in Figure 2.1b.

Lemma 1. *Consider $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ be the input automaton derived from $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$. The policies we have are as follows:*

1 $\forall (x,y) \in \Sigma, \forall q, q' \in Q : q \xrightarrow{(x,y)} q' \implies q \xrightarrow{x}_I q'$.
2 $\forall x \in \Sigma_I, \forall q, q' \in Q : q \xrightarrow{x}_I q' \implies \exists y \in \Sigma_O : q \xrightarrow{(x,y)} q'$.

Lemma 1 is an immediate consequence from Definitions 1 and 2. Policy 1 states that if there is a transition from state $q \in Q$ to state $q' \in Q$ in the automaton $\mathcal{A}_\varphi$ upon input-output event $(x, y) \in \Sigma$, then there is a transition from state q to state q' in the input automaton $\mathcal{A}_{\varphi_I}$ upon the input event $x \in \Sigma_I$. Policy 2 states that if there is a transition from state $q \in Q$ to state $q' \in Q$ upon input event $x \in \Sigma_I$, then there must be an output event $y \in \Sigma_O$ s.t. there is a transition from state q to state q' upon event (x, y) in the automaton $\mathcal{A}_\varphi$.

Definition 3 (Product of SA)**.** *Given two SA $\mathcal{A}_{\varphi_1} = (Q^1, q_0^1, q_v^1, \Sigma, \rightarrow_1)$, and $\mathcal{A}_{\varphi_2} = (Q^2, q_0^2, q_v^2, \Sigma, \rightarrow_2)$, their product SA $\mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2} = (Q, q_0, q_v, \Sigma, \rightarrow)$ where $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, $q_v = (q_v^1, q_v^2)$, and the transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$ with $((q^1, q^2), a, (q'^1, q'^2)) \in \rightarrow$ if $(q^1, a, q'^1) \in \rightarrow^1$ and $(q^2, a, q'^2) \in \rightarrow_2$.*

In the product SA $\mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2}$, all the locations in $(Q^1 \times q_v^2) \cup (q_v^1 \times Q^2)$ are trap locations. All the outgoing transitions from these locations can be replaced with self-loops, and all such locations can be merged into a single violating location labeled as q_v. Any outgoing transition from a location in $Q \setminus (Q^1 \times q_v^2) \cup (q_v^1 \times Q^2)$ to a location in $(Q^1 \times q_v^2) \cup (q_v^1 \times Q^2)$ goes to q_v instead.

The product of SAs is useful to enforce multiple policies using the monolithic approach by first constructing a product of the given SAs. Given two deterministic and

complete SAs $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$, the product SA $\mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2}$ is deterministic and complete which recognizes the language $\mathcal{L}(\mathcal{A}_{\varphi_2}) \cap \mathcal{L}(\mathcal{A}_{\varphi_2})$.

2.1.1 Edit Functions

Let us consider policy $\varphi \subseteq \Sigma^*$, specified as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, and SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ derived from $\mathcal{A}_\varphi$ by discarding outputs. The enforcer utilizes the following $\mathsf{editI}_{\varphi_\mathsf{I}}$ (resp. editO_φ), for editing input (resp. output) events (when required), as per the policy φ_I (resp. φ).

- **$\mathsf{editI}_{\varphi_\mathsf{I}}(\sigma_I)$**: Given $\sigma_I \in \Sigma_I^*$, $\mathsf{editI}_{\varphi_\mathsf{I}}(\sigma_I)$ is the set of input events $x \in \Sigma_I$ s.t. the word obtained by concatenating x after σ_I satisfies policy φ_I. Formally,

$$\mathsf{editI}_{\varphi_\mathsf{I}}(\sigma_I) = \{x \in \Sigma_I : \sigma_I \cdot x \models \varphi_I\}.$$

  When we consider the SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$, the members in Σ_I that allow to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q \setminus \{q_v\}$ is defined as:

$$\mathsf{editI}_{\mathcal{A}_{\varphi_\mathsf{I}}}(q) = \{x \in \Sigma_I : q \xrightarrow{x}_I q' \wedge q' \neq q_v\}.$$

  Let us, for example, consider the SA in Figure 2.1b derived from the SA in Figure 2.1a by projecting on inputs. If we consider $\sigma = (10, 0) \cdot (01, 1)$, we have $\sigma_I = 10 \cdot 01$. Then, $\mathsf{editI}_{\varphi_\mathsf{I}}(\sigma_I) = \Sigma_I \setminus \{11\}$. Moreover, $q_0 \xrightarrow{10\cdot01}_I q_0$, and $\mathsf{editI}_{\mathcal{A}_{\varphi_\mathsf{I}}}(q_0) = \Sigma_I \setminus \{11\}$. If $\mathsf{editI}_{\mathcal{A}_{\varphi_\mathsf{I}}}(q)$ is non-empty, then $\mathsf{nondet} - \mathsf{editI}_{\mathcal{A}_{\varphi_\mathsf{I}}}(q)$ returns an element (chosen randomly) from $\mathsf{editI}_{\mathcal{A}_{\varphi_\mathsf{I}}}(q)$ and is undefined if $\mathsf{editI}_{\mathcal{A}_{\varphi_\mathsf{I}}}(q)$ is empty.
- **$\mathsf{editO}_\varphi(\sigma, x)$**: Consider an input event $x \in \Sigma_I$, and an input-output word $\sigma \in \Sigma^*$. We have $\mathsf{editO}_\varphi(\sigma, x)$, the set of output events y in Σ_O s.t. the input-output word obtained by concatenating σ followed by (x, y) (i.e., $\sigma \cdot (x, y)$) satisfies policy φ. Formally,

$$\mathsf{editO}_\varphi(\sigma, x) = \{y \in \Sigma_O : \sigma \cdot (x, y) \models \varphi\}.$$

  When we consider the automaton $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$ specifying policy φ, and an

input event $x \in \Sigma_I$, the set of output events y in Σ_O permitting to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q \setminus \{q_v\}$ with (x, y) is defined as:

$$\mathsf{editO}_{\mathcal{A}_\varphi}(q, x) = \{y \in \Sigma_O : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v\}.$$

For example, consider policy P defined by the automaton in Figure 2.1a. We have $\mathsf{editO}_{\mathcal{A}_\varphi}(q_0, 01) = \{0\}$.

If $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$ is not empty, then $\mathsf{nondet} - \mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$ returns a random element from $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$, and if $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$ is empty $\mathsf{nondet} - \mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$ is undefined.

2.1.2 Runtime Enforcement for Synchronous Programs

In this section, we briefly recall the RE problem for synchronous programs from [22]. In this setting, that we also consider in this work, as illustrated in Figure **??**, an enforcer monitors and corrects both inputs and outputs of a synchronous program according to a given safety policy $\varphi \subseteq \Sigma^*$.

The model hypothesizes that the black-box synchronous program can be called using a custom function call called *ptick* that is called just once during each reaction / synchronous step. We can formally consider ptick as a function from Σ_I to Σ_O that accepts a bit vector $x \in \Sigma_I$ and returns a bit vector $y \in \Sigma_O$.

An enforcer for the policy φ can only alter an input-output event when it's absolutely essential; it can't block, postpone, or suppress events. Let's remember the two functions $\mathsf{editI}_{\varphi_\mathsf{I}}$ and editO_φ from Section **??**, which the enforcer for φ uses to edit the current input (or output) event according to the policy φ. An enforcer may be thought of as a function that modifies input-output words at a high level. An enforcement function for the policy φ takes an input-output word over Σ as input and produces an input-output word over Σ that conforms to φ as output.

We reproduce from [22] and briefly discuss, Definition 4 of the constraints that an

enforcer for any given policy φ should satisfy:

Definition 4 (Enforcer for φ)**.** *An* enforcer *for a given policy $\varphi \subseteq \Sigma^*$ is a function $E_\varphi : \Sigma^* \to \Sigma^*$ satisfying the following constraints:*

Soundness

$$\forall \sigma \in \Sigma^* : E_\varphi(\sigma) \models \varphi. \tag{\textbf{Snd}}$$

Monotonicity

$$\forall \sigma, \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \Rightarrow E_\varphi(\sigma) \preccurlyeq E_\varphi(\sigma'). \tag{\textbf{Mono}}$$

Instantaneity

$$\forall \sigma \in \Sigma^* : |\sigma| = |E_\varphi(\sigma)|. \tag{\textbf{Inst}}$$

Transparency

$$\begin{aligned} &\forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O : \\ &\quad E_\varphi(\sigma) \cdot (x, y) \models \varphi \implies \\ &E_\varphi(\sigma \cdot (x, y)) = E_\varphi(\sigma) \cdot (x, y). \end{aligned} \tag{\textbf{Tr}}$$

Causality

$$\begin{aligned} &\forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O, \exists x' \in \mathsf{editI}_{\varphi_\mathsf{I}}(E_\varphi(\sigma)_I), \\ &\quad \exists y' \in \mathsf{editO}_\varphi(E_\varphi(\sigma), x') : E_\varphi(\sigma \cdot (x, y)) = \\ &E_\varphi(\sigma) \cdot (x', y'). \end{aligned} \tag{\textbf{Cau}}$$

The enforcer releases the input-output sequence $E_\varphi(\sigma)$ as output after reading the input-output sequence σ, and $E_\varphi(\sigma)_I \in \Sigma_I^*$ is the projection on the inputs. Note that $\mathsf{editI}_{\varphi_\mathsf{I}}(E_\varphi(\sigma)_I)$ returns a set of input events in Σ_I, s.t. $E_\varphi(\sigma)_I$ (which is the input alphabet projection of the input-output word $E_\varphi(\sigma)$) followed by any event from $\mathsf{editI}_{\varphi_\mathsf{I}}(E_\varphi(\sigma)_I)$ satisfies φ_I. $\mathsf{editO}_\varphi(E_\varphi(\sigma), x')$ returns a set of output events in Σ_O, such that for any event y in $\mathsf{editO}_\varphi(E_\varphi(\sigma), x')$, $E_\varphi(\sigma) \cdot (x', y)$ satisfies φ.

- Soundness (**Snd**) states that the output of the enforcer $E_\varphi(\sigma)$ must satisfy φ for any word $\sigma \in \Sigma^*$.
- Monotonicity (**Mono**) specifies that the enforcer's output for an extended word σ' of a word σ extends the enforcer's output for σ. The enforcer cannot undo what has

already been transmitted as output due to the monotonicity condition.

- Instantainety (**Inst**) states that for any given input-output word σ, the enforcer's output $E_\varphi(\sigma)$ should contain exactly the same number of events as σ (i.e., E_φ is length-preserving). As a result, the enforcer is unable to delay, insert, or suppress events. When the enforcer receives a new event, it must respond immediately and provide an output event instantaneously.
- Transparency (**Tr**) states that for any given word σ and event (x, y), if the enforcer's output for σ (i.e., $E_\varphi(\sigma)$) followed by the event (x, y) fulfills the policy φ (i.e., $E_\varphi(\sigma) \cdot (x, y) \models \varphi$), then the output that the enforcer produces for input $\sigma \cdot (x, y)$ will be $E_\varphi(\sigma) \cdot (x, y)$. This means that when no modification is required to meet the policy φ, the enforcer does nothing.
- Causality (**Cau**) states that the enforcer generates input-output event (x', y') for every input-output event (x, y), where the enforcer first processes the input portion x to produce the transformed input x' according to policy φ using $\mathsf{editI}_{\varphi_\mathsf{I}}$ for every input-output event (x, y). After executing function ptick with the transformed input x', the enforcer reads and transforms output $y \in \Sigma_O$, which is the program's output, to generate the transformed output y' using editO_φ.

Remark 2. *After reading input-output sequence $\sigma \in \Sigma^*$, let $E_\varphi(\sigma)$ be the input-output sequence produced as output by the enforcer for φ. If what has already been computed as output by the enforcer $E_\varphi(\sigma)$ followed by (x, y) does not allow to satisfy the policy φ, the enforcer edits (x, y) using functions $\mathsf{editI}_{\varphi_\mathsf{I}}$ and editO_φ when reading a new event (x, y). When editing the current event (x, y), important to note that there may be several options.*

Let us consider the policy P from Example 1. If we consider $\sigma = (10, 1) \cdot (01, 0)$, the output of the enforcer after reading σ should be $E_\varphi(\sigma) = (10, 1) \cdot (01, 0)$. Consider another new event $(11, 0)$, and $E_\varphi(\sigma) \cdot (11, 0)$ does not satisfy φ, and the enforcer thus has to alter this new event $(11, 0)$. We have $E_\varphi(\sigma)_I = 10 \cdot 01$, and since $\mathsf{editI}_{\varphi_\mathsf{I}}(10 \cdot 01) = \{00, 01, 10\}$ the enforcer can choose any element from this set as the transformed input.

Definition 5 (Enforceability)**.** *Let $\varphi \subseteq \Sigma^*$ be a policy. We say that φ is* enforceable

iff an enforcer E_φ for φ exists according to Definition 4.

Remark 3 (Not all safety policies are enforceable)**.** *Not all policies are enforceable, even if we restrict ourselves to prefix-closed safety policies as is shown and illustrated in [22].*

Remark 4 (Condition for enforceability)**.** *We recall the enforceability condition proposed and proved in [22]. Consider a policy φ defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$. Policy φ is enforceable iff the following condition holds:*

$$\forall q \in Q, q \neq q_v \implies \exists (x,y) \in \Sigma : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v \qquad \textbf{(EnfCo)}$$

It's worth noting that given any policy φ defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, to test whether $\mathcal{A}_\varphi$ satisfies condition (**EnfCo**) *is straightforward.*

2.2 Runtime Enforcement Framework for Policies Defined as SA

In this section, we recall the definition of an enforcement function from [22], which incrementally builds the output and presents how any given word $\sigma \in \Sigma^*$ is transformed according to the policy φ.

A pair (x, y) is an input-output event (reaction), where $x \in \Sigma_I$ is the input, and $y \in \Sigma_O$ is the output. The enforcer immediately produces an input-output event (x', y') as output after receiving an input-output event (x, y) as input. The enforcer processes the input x first, producing a transformed input x', and then the output y, producing the transformed event (x', y'). The enforcement function E_φ is made up of two functions: E_I and E_O. E_I reads the input x (from the environment) and produces a transformed input x', while E_O reads the transformed input x' (output of E_I) and the output y (which is the output obtained by invoking ptick with x') and adds the transformed event (x', y') to the output of the enforcer.

Definition 6 (Enforcement function)**.** *The enforcement function* $E_\varphi : \Sigma^* \to \Sigma^*$ *for a given policy* $\varphi \subseteq \Sigma^*$*, is defined as* $E_O(E_I(\sigma_I), \sigma_O)$*:*

where:

- $E_I : \Sigma_I^* \to \Sigma_I^*$ *is defined as:*

$$
\begin{aligned}
E_I(\epsilon_{\Sigma_I}) &= \epsilon_{\Sigma_I} \\
E_I(\sigma_I \cdot x) &= \begin{cases} E_I(\sigma_I) \cdot x & \text{if } E_I(\sigma_I) \cdot x \models \varphi_I, \\ E_I(\sigma_I) \cdot x' & \text{otherwise} \end{cases}
\end{aligned}
$$

  where $x' = \mathsf{nondet} - \mathsf{editI}_{\varphi_\mathsf{I}}(E_I(\sigma_I))$.

- $E_O : \Sigma_I^* \times \Sigma_O^* \to (\Sigma_I \times \Sigma_O)^*$ *is defined as:*

$$
\begin{aligned}
E_O(\epsilon_{\Sigma_I}, \epsilon_{\Sigma_O}) &= \epsilon_\Sigma \\
E_O(\sigma_I \cdot x, \sigma_O \cdot y) &= \begin{cases} E_O(\sigma_I, \sigma_O) \cdot (x, y) & \text{if} \\ \qquad\qquad E_O(\sigma_I, \sigma_O) \cdot (x, y) \models \varphi, \\ E_O(\sigma_I, \sigma_O) \cdot (x, y') & \text{otherwise} \end{cases}
\end{aligned}
$$

  where $y' = \mathsf{nondet} - \mathsf{editO}_\varphi(E_O(\sigma_I, \sigma_O), x)$.

The function E_φ accepts a word over Σ^* and outputs another word over Σ^*. We have $\sigma_I \in \Sigma_I^*$ is the projection of σ on inputs, and $\sigma_O \in \Sigma_O^*$ is the projection of σ on outputs, for a word $\sigma \in \Sigma^*$. The result of function E_O is the output of the enforcement function E_φ, which is defined through two functions, E_I and E_O.

Function E_I*:* Function E_I accepts the word obtained by projecting on the inputs ($\sigma_I \in \Sigma_I^*$) as input and returns a word in Σ_I^* as output for a given word $\sigma \in \Sigma^*$. Inductively, the function E_I is defined. When the input $\sigma_I = \epsilon_{\Sigma_I}$, it returns$\epsilon_{\Sigma_I}$. When Σ_I is read as input and $E_I(\sigma_I)$ is returned as output, there are two possible possibilities depending on whether $E_I(\sigma_I) \cdot x$ fulfills the policy φ_I or not.

- If $E_I(\sigma_I)$ succeeded by the new input x satisfies the input policy φ_I, then the new input x is concatenated to the previous output of function E_I (that is, $E_I(\sigma_I \cdot x) = E_I(\sigma_I) \cdot x$).
- Otherwise, $E_I(\sigma_I) \cdot x$ does not satisfy φ_I. In this case, input x is converted using $\mathsf{nondet} - \mathsf{editI}_{\varphi_\mathsf{I}}(E_I(\sigma_I))$ to obtain transformed input x', which is appended to the previous output of function E_I (that is, $E_I(\sigma_I \cdot x) = E_I(\sigma_I) \cdot x'$). $\mathsf{nondet} - \mathsf{editI}_{\varphi_\mathsf{I}}(E_I(\sigma_I))$ returns $x' \in \Sigma_I$, such that φ_I is satisfied by the preceding output of function E_I followed by x'.

Function E_O: Function E_O takes an input word from Σ_I^* and an output word from Σ_O* as input and returns an input-output word in Σ^*, which is a sequence of tuples with an input and an output for each event. Inductively, the function E_O is defined. The output of E_O is ϵ when both the input and output words are empty. If $\sigma_I \in \Sigma_I^*$ and $\sigma_O \in \Sigma_O^*$ is read, the output will be $E_O(\sigma_I, \sigma_O)$, and if another fresh input event x and output event y are observed, there are two alternatives depending on whether $E_O(\sigma_I, \sigma_O) \cdot (x, y)$ satisfies φ or not.

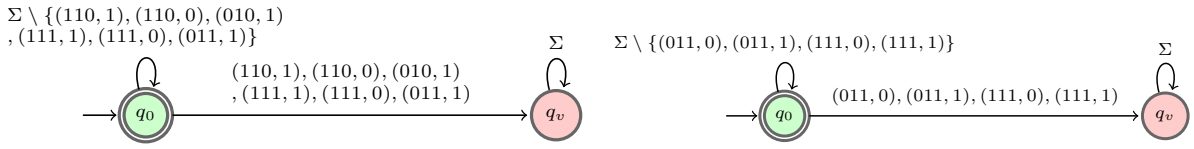
- If $E_O(\sigma_I, \sigma_O)$ succeeded by (x, y) respects φ, then (x, y) is added to the previous output of function E_O (i.e., $E_O(\sigma_I \cdot x, \sigma_O \cdot y) = E_O(\sigma_I, \sigma_O) \cdot (x, y)$).
- If the preceding case is not satisfied, then $E_O(\sigma_I, \sigma_O) \cdot (x, y)$ does not respect/satisfy φ. $\mathsf{nondet} - \mathsf{editO}_\varphi(E_O(\sigma_I, \sigma_O), x)$ is thus used to alter output y to obtain y' (altered output), and the event (x, y') is added to the previous output of the function E_O (i.e., $E_O(\sigma_I \cdot x, \sigma_O \cdot y) = E_O(\sigma_I, \sigma_O) \cdot (x, y')$). $\mathsf{nondet} - \mathsf{editO}_\varphi(E_O(\sigma_I, \sigma_O), x)$ outputs $y' \in \Sigma_O$ such that φ is satisfied by the preceding output of function E_O followed by (x, y').

Remark 5 (Functional definition satisfies constraints)**.** *In [22], it is proved that for any given policy φ that is enforceable, the enforcer defined as function E_φ (Definition 6) satisfies the* (**Snd**)*,* (**Tr**)*,* (**Mono**)*,* (**Inst**)*, and* (**Cau**) *constraints (Definition* 4*).*

Example 2 (Functional definition)**.** *Let us consider the policy "B and Q can't happen at the same time, and Q and X can't happen at the same time" illustrated in Figure 2.1a,*

σ_I	σ_O	$E_I(\sigma_I)$	$E_\varphi(\sigma) = E_O(E_I(\sigma_I), \sigma_O)$
ϵ_I	ϵ_O	ϵ_I	$(\epsilon_I, \epsilon_O) = \epsilon$
01	0	01	$(01, 0)$
$01 \cdot 01$	$0 \cdot 1$	$01 \cdot 01$	$(01, 0) \cdot (01, \mathbf{0})$

Table 2.1: Functional definition example



(a) SA $\mathcal{A}_{S_1}$. (b) SA $\mathcal{A}_{S_2}$.

Figure 2.2: Safety automaton for S_1 and S_2.

where $I = \{B, Q\}$ *and* $O = \{X\}$. *The output of functions* E_I, E_O *is illustrated in Table 2.1 when the input sequence* $\sigma = (01, 0) \cdot (01, 1)$ *(where* $\sigma_I = 01 \cdot 01$ *and* $\sigma_O = 0 \cdot 1$*) is processed incrementally by the enforcement function. When* σ *is (01,0), since it satisfies policy* P*, it is emitted without any alteration. For the second event (01,1) (*$\sigma = (01, 0) \cdot (01, 1)$*), the input enforcer* $E_I(\sigma_I) = 01 \cdot 01$ *since it does not violate the input policy but since the output of 1 in this step violates the policy* P*; it is transformed into 0 satisfying the policy. Thus, the enforcer outputs* $(01, 0) \cdot (01, \boldsymbol{0})$.

2.3 Monolithic and Incremental Schemes for Enforcing Multiple Policies

In this section, we focus on the problem of how we enforce a given set of policies expressed as SA in the considered reactive systems framework.

Example 3 (Example policies)**.** *Let* $I = \{A, B, C\}$ *and* $O = \{R\}$. *Consider the following policies:* S_1*: "A and B cannot happen simultaneously, and also B and R cannot happen simultaneously" and* S_2*: "B and C cannot happen simultaneously". The safety automaton in Figure 2.6a and Figure 2.6b define policies* S_1 *and* S_2 *respectively.*

σ_I	σ_O	$E_I(\sigma_I)$	$E_\varphi(\sigma) = E_O(E_I(\sigma_I), \sigma_O)$
ϵ_I	ϵ_O	ϵ_I	$(\epsilon_I, \epsilon_O) = \epsilon$
100	1	100	$(100, 1)$
$100 \cdot 110$	$1 \cdot 1$	$100 \cdot 100$	$(100, 1) \cdot (100, 1)$
$100 \cdot 110 \cdot 011$	$1 \cdot 1 \cdot 0$	$100 \cdot 100 \cdot 001$	$(100, 1) \cdot (100, 1) \cdot (001, 0)$

Table 2.2: Example illustrating behavior of enforcer for $\mathcal{A}_{S_1 \cap S_2}$

2.3.1 Monolithic security enforcement

Composing all of the policies first is one way to enforce a collection of policies (taking the product of all the SA). We can synthesize one enforcer for the resulting policy if the resulting SA is enforceable according to Definition 5.

In the monolithic approach, policies (specified as SA) are first combined using intersection (see the Definition 3, the product of SA), and an enforcer for the resulting policy is synthesized. Specifically, given any two safety policies φ_1 and φ_2, to enforce both these policies, we first compute $\varphi = \varphi_1 \cap \varphi_2$ (by computing the product of SA for φ_1 and φ_2). Then if the resulting SA for φ is enforceable as per Definition 5, we synthesize an enforcer for φ using the approach described in Section 2.2.

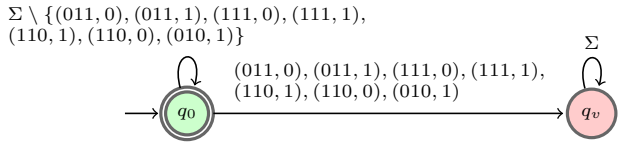


Figure 2.3: $\mathcal{A}_{S_1 \cap S_2}$: Product of Automaton S_1 and S_2

Example 4 (Monolithic approach)**.** *Consider policies S_1 and S_2 defined as SAs illustrated in Figure 2.2. The SA obtained by taking the product of both these automata is shown in Figure 2.3 defining the policy $S_1 \cap S_2$. The policy $S_1 \cap S_2$ is enforceable since for every accepting state, there is at least one outgoing transition to an accepting state (See Remark 4). Table 2.2 illustrates behavior of enforcer for policy $\mathcal{A}_{S_1 \cap S_2}$ when the input-output word $(100, 1) \cdot (110, 1) \cdot (011, 0)$ is processed incrementally.*

Theorem 2 (Enforceability using the monolithic approach)**.** *Consider two policies φ_1, φ_2 defined as SA, and $\varphi = \varphi_1 \cap \varphi_2$.*

If policy φ_1 or policy φ_2 is non-enforceable, then $\varphi_1 \cap \varphi_2$ is non-enforceable.

The proof of Theorem 2 is given in Appendix **??**.

Remark 6 (Enforceability using the monolithic approach)**.** *Though policies φ_1 and φ_2 are enforceable individually, policy $\varphi_1 \cap \varphi_2$ may not be enforceable, as illustrated in the following example.*

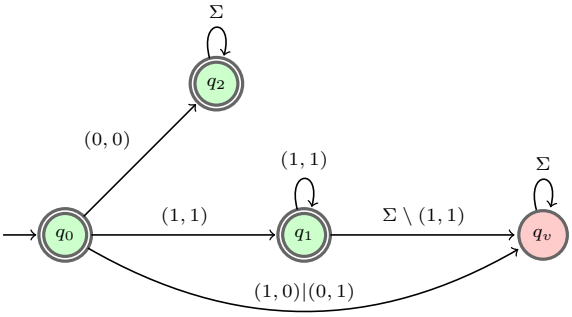
Example 5 (Monolithic approach does not always work)**.** *Consider the two policies shown in Figure 2.4. Here $I = \{0, 1\}$ and $O = \{0, 1\}$. Though they are enforceable individually, the policy that we obtain by taking the product of both the SA $\varphi_1 \cap \varphi_2$ is not enforceable. Suppose that the first event is $(1, 1)$, the output of the enforcer will be $(1, 1)$, and the state of the product automaton will be updated to (q_1, l_1). When in state (q_1, l_1), whatever may be the input event, it is not possible to correct it (as there will be no path to an accepting state from the state (q_1, l_1) in the product automaton).*

As discussed in the problem description in the introduction, we focus on how to add a new enforcement layer incrementally to the existing enforcer (e.g., when a new property to be enforced is identified due to a new threat). How can an enforcer for the new property be composed with an existing enforcer such that the composed enforcement system, along with the new policy, will also continue to enforce all the previously enforced policies is what we explore and study in this work.

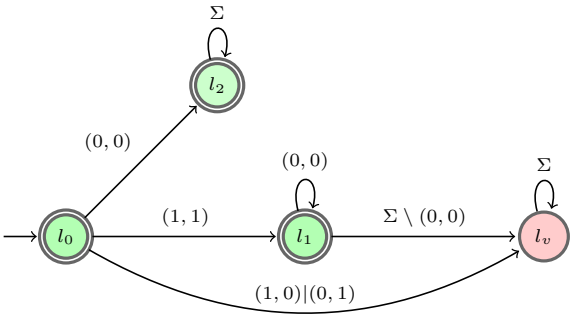
In our framework, since the framework allows editing of events, and since enforcers are bi-directional, it is not possible to compose enforcers (as per Definition 6) for the following reasons:

- As illustrated in Figure **??**, the enforcer is bi-directional. Firstly, input from the environment is read (and edited/corrected if necessary), which is fed to the program, and the resulting output of the program is later checked (and edited if necessary) by the enforcer before it is forwarded to the environment. This does not allow the enforcers that we have to be composed directly in series.

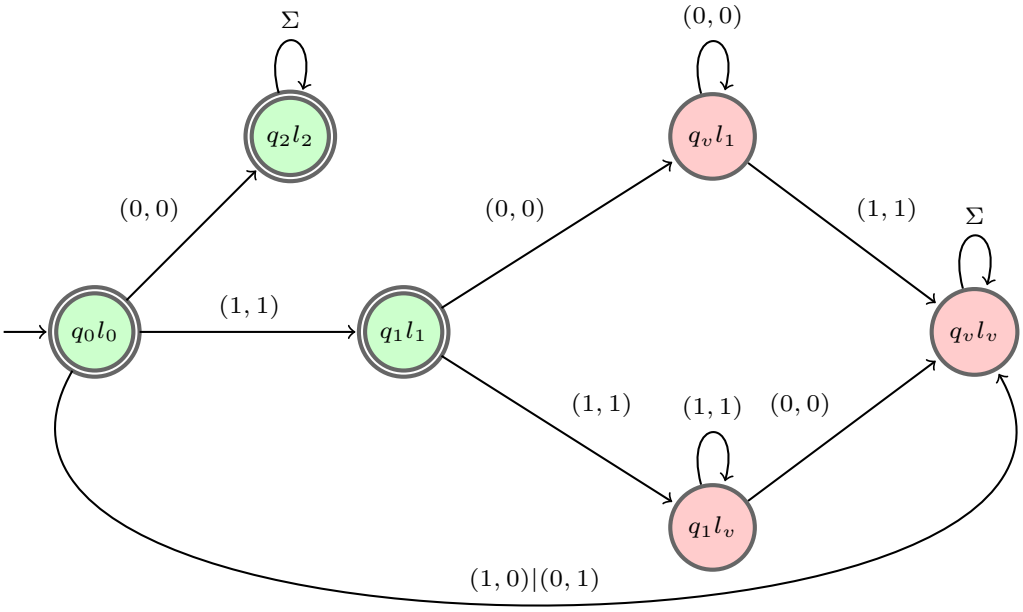
[2]Here, $\Sigma = \{(0,0), (0,1), (1,0), (1,1)\}$. So $\Sigma \setminus (1,1) = \{(0,0), (0,1), (1,0)\}$.
[3]Here, $\Sigma = \{(0,0), (0,1), (1,0), (1,1)\}$. So $\Sigma \setminus (0,0) = \{(0,1), (1,0), (1,1)\}$.



(a) SA defining φ_1- Enforceable [2]



(b) SA defining φ_2- Enforceable [3]



(c) SA defining $\varphi_1 \cap \varphi_2$

Figure 2.4: Policy $\varphi_1 \cap \varphi_2$ is non-enforceable.

- Moreover, since editing of events is allowed, the edit/correction made by one enforcer may not be compatible w.r.t the other enforcer, where there may be some edit/correction that is suitable for both policies to be enforced.

Thus we need to revisit the definition of the enforcement function, which will also be suitable for incremental enforcement schemes.

2.3.2 Incremental composition of security enforcers

Suppose that we rely on the internals of the enforcement function, which composes an input enforcement function and an output enforcement function, then we can consider:

- composing all the input enforcement functions in series, where the (corrected) input that is released by the last function is fed to the program.
- similarly, the output enforcement functions can also be combined in series, and the corrected output released by the last function can be emitted to the environment.

The incremental scheme by composing enforcers in series is shown in Figure 2.5.

Given two policies φ_1 and φ_2 (where φ_{1I} and φ_{2I} are their corresponding input policies), we can synthesize input and output enforcement functions for each of these policies $E_I\varphi_1$, $E_I\varphi_2$, $E_O\varphi_1$, and $E_O\varphi_2$, and then compose all the input enforcers and all the output enforcers. The composed input enforcer can be then combined with the composed output enforcer (see Figure 2.5).

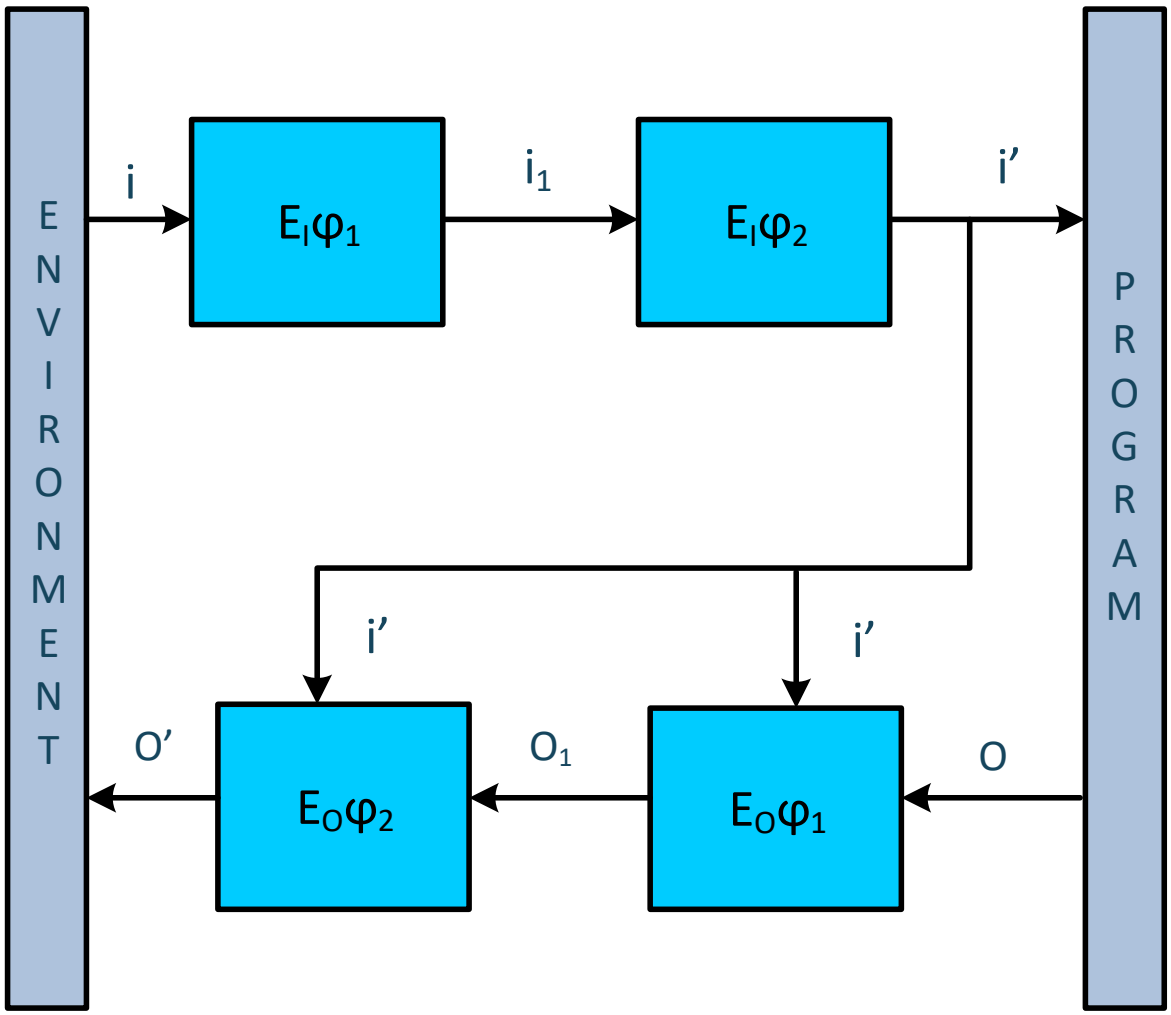


Figure 2.5: Incremental enforcement via serial composition

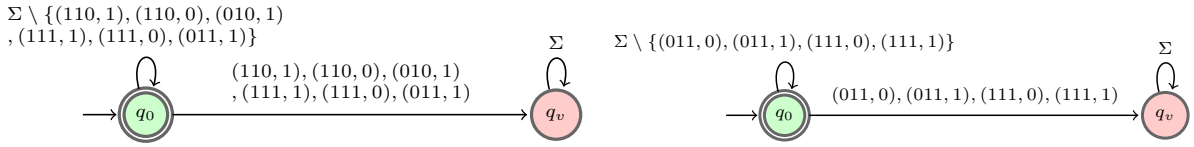
We denote this type of incremental composition of enforcers in series as $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$. In this section we investigate whether $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$ generally enforces $\varphi_1 \cap \varphi_2$. We are also interested to see whether the final output that we obtain using the incremental composition approach is equal to the output we would obtain using the monolithic approach.

Let us now formally define incremental composition in series of two enforcers.

Definition 7 (Incremental enforcement via serial composition)**.** *Let $E_I\varphi_1 : \Sigma_I^* \to \Sigma_I^*$ (resp. $E_I\varphi_2 : \Sigma_I^* \to \Sigma_I^*$) be the input enforcement function for policy $\varphi_{1I} \subseteq \Sigma_I^*$ (resp. φ_{2I}), and let $E_O\varphi_1 : \Sigma_I^* \times \Sigma_O^* \to (\Sigma_I \times \Sigma_O)^*$ (resp. $E_O\varphi_2$) be the output enforcement function for policy $\varphi_1 \subseteq \Sigma^*$ (resp. φ_2). Their serial composition is a new enforcer*

σ_I	σ_O	$\sigma_I' = E_I S1(\sigma_I)$	$\sigma_I'' = E_I S2(\sigma_I')$	$\sigma_O' = E_O S1(\sigma_I'', \sigma_O)$	$E_O S2(\sigma_I'', \sigma_O')$
100	1	100	100	$(100, 1)$	$(100, 1)$
$100 \cdot 110$	$1 \cdot 1$	$100 \cdot 011$	$100 \cdot 110$	$(100, 1) \cdot ??$	–

Table 2.3: Serial composition using Definition 7



(a) SA $\mathcal{A}_{S_1}$. (b) SA $\mathcal{A}_{S_2}$.

Figure 2.6: Safety automaton for S_1 and S_2.

$E_{\varphi_1} \Rrightarrow E_{\varphi_2} : \Sigma^* \to \Sigma^*$ *defined as follows:*

$$\forall \sigma \in \Sigma^*, (E_{\varphi_1} \Rrightarrow E_{\varphi_2})(\sigma) = E_O\varphi_2(\sigma_I', \sigma_O').$$

with $\sigma_I' = E_I\varphi_2(E_I\varphi_1(\sigma_I))$, *and* $\sigma_O' = E_O\varphi_1(\sigma_I', \sigma_O)$.

As per the composition Definition 7, the output of the serial composition of the enforcers is the output emitted from the output enforcer $E_O\varphi_2$. The input emitted from the input enforcer $E_I\varphi_2$ is considered as the final corrected input. The output enforcer $E_O\varphi_1$ is invoked with the corrected input σ_I' from the serially composed input enforcers and the output σ_O of the reactive system. The corrected output of the enforcer is input to the output enforcer $E_O\varphi_2$, which finally emits the output to the environment.

Remark 7. *Definition 7 is formulated in order to support incrementally adding a new enforcement layer. Suppose that we only have the input and output enforcement functions w.r.t policy* φ_1 *and a new policy to be enforced* φ_2 *is given. We can obtain enforcement functions for* φ_2 *individually and compose with the existing enforcement functions as per Definition 7.*

Note that serial composition of enforcers as per Definition 7 does not always work. That is, though given two policies φ_1, φ_2 *and also* $\varphi_1 \cap \varphi_2$ *are all enforceable, the serial composition of enforcers of* φ_1 *and* φ_2 *as per the above definition may not work. The final output obtained may not satisfy* $\varphi_1 \cap \varphi_2$. *Moreover, there may also be situations*

where other constraints, such as instantaneity, may be violated. Let us consider input enforcement to understand this (similar reasoning also applies for output enforcement). As per the serial composition definition (Def. 7), the input emitted from the input enforcer $E_I\varphi_2$ is considered as the final corrected input, but the final input selected by the function $E_I\varphi_2$ may violate the policy monitored by the input enforcer $E_I\varphi_1$.

Let us consider the following example to understand this further.

Example 6 (Serial composition as per Definition 7 does not always work)**.** *Let us again consider policies S_1 and S_2 illustrated in Figure 2.2, presented again in Figure 2.6. Both policies S_1 and S_2 are enforceable individually. The policy $S_1 \cap S_2$ is also enforceable. However, when we compose input and output enforcers for these policies in series as per Definition 7, the final output obtained may not satisfy policy $S_1 \cap S_2$. Also, there may be situations where constraints such as instantaneity may be violated. For example, consider the word $(100, 1) \cdot (110, 1) \cdot (011, 0)$ to be processed incrementally. In the first step, $(100, 1)$ satisfies both policies and thus will be emitted as it is. In the second step, let us consider that the altered input produced by the second input enforcement function is 110. When 110 is fed as input to the output enforcement function of policy S_1, there is no possible output event that it can release to satisfy policy S_1. The incremental processing of the considered word is shown in Table 2.3.*

In this section, we have seen that the enforcement function in Definition 6 is not always suitable for the incremental security enforcement by composing enforcers in series, also when we consider first composing all the input enforcement functions, followed by the composition of the output enforcement functions (Definition 7).

We thus revisit the incremental security enforcement scheme in the next section, to propose an incremental scheme that can tackle all the enforceable properties.

2.4 Revisiting Incremental Security Enforcement Scheme

In this section we propose an incremental composition scheme. First, we define the following Select functions and later present how the compositional scheme can be defined using the Select functions.

2.4.1 Select Functions

- **$\mathsf{SelectI}_{\varphi_I}(\sigma_I, X)$**: Given an input word $\sigma_I \in \Sigma_I^*$, and a set of input events $X \subseteq \Sigma_I$, $\mathsf{SelectI}_{\varphi_I}(\sigma_I, X)$ is the set of input events x that belong to set X such that the word obtained by extending σ_I with x satisfies policy φ_I. Formally,

$$\mathsf{SelectI}_{\varphi_I}(\sigma_I, X) = \{x \in X : \sigma_I \cdot x \models \varphi_I\}.$$

  Considering the SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$, the set of events in X that allow to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q \setminus \{q_v\}$ is defined as:

$$\mathsf{SelectI}_{\mathcal{A}_{\varphi_I}}(q, X) = \{x \in X : q \xrightarrow{x}_I q' \wedge q' \neq q_v\}.$$

  For example, let us consider the input automaton corresponding to policy P in Figure 2.1b. Initially, when $\sigma_I = \epsilon$ we have $X = \{00, 01, 10, 11\}$, and $\mathsf{SelectI}_P(\epsilon, X) = \{00, 01, 10\}$. If we consider $\sigma_I = 00{\cdot}01{\cdot}01$, and $X = \{00, 01, 10\}$, we have $\mathsf{SelectI}_P(00{\cdot}01, X) = \{00, 01, 10\}$.
  Also, $q_0 \xrightarrow{00\cdot01}_I q_0$, and $\mathsf{SelectI}_P(q_0, \{00, 01, 10\}) = \{00, 01, 10\}$.
- **$\mathsf{SelectO}_{\varphi}(\sigma, x, Y)$**: Given an input-output word $\sigma \in \Sigma^*$, an input event $x \in \Sigma_I$, and a set of output events $Y \subseteq \Sigma_O$, $\mathsf{SelectO}_{\varphi}(\sigma, x, Y)$ is the set of output events y in Y s.t. the input-output word obtained by extending σ with (x, y) satisfies policy φ. Formally,

$$\mathsf{SelectO}_{\varphi}(\sigma, x, Y) = \{y \in Y : \sigma \cdot (x, y) \models \varphi\}.$$

  Considering the automaton $\mathcal{A}_{\varphi} = (Q, q_0, q_v, \Sigma, \rightarrow)$ defining policy φ, and an input

σ_I	σ_O	$X_1 = \mathsf{SelectI}_{S1}(\sigma'_I, \Sigma_I)$	$X' = \mathsf{SelectI}_{S2}(\sigma'_I, X_1)$	$x' = MinD(x, X')$	σ'_I
100	1	$\mathsf{SelectI}_{S1}(\epsilon_I, \Sigma_I) = \{100, 101, 010, 001, 011\}$	$\mathsf{SelectI}_{S2}(\epsilon_I, X_1) = \{100, 101, 010, 001\}$	$MinD(100, \{100, 101, 010, 001\}) = 100$	100
$100 \cdot 110$	$1 \cdot 1$	$\mathsf{SelectI}_{S1}(100, \Sigma_I) = \{100, 101, 010, 001, 011\}$	$\mathsf{SelectI}_{S2}(100, X_1) = \{100, 101, 010, 001\}$	$MinD(110, \{100, 101, 010, 001\}) = 100$	$100 \cdot 100$
$100 \cdot 110 \cdot 011$	$1 \cdot 1 \cdot 0$	$\mathsf{SelectI}_{S1}(100.100, \Sigma_I) = \{100, 101, 010, 001, 011\}$	$\mathsf{SelectI}_{S2}(100 \cdot 100, X_1) = \{100, 101, 010, 001\}$	$MinD(011, \{100, 101, 010, 001\}) = 001$	$100 \cdot 100 \cdot 001$

Table 2.4: Serial composition scheme using Select()-input enforcement

$Y' = \mathsf{SelectO}_{S1}(\sigma', x', \Sigma_O)$	$Y'' = \mathsf{SelectO}_{S2}(\sigma', x', Y')$	$MinD(y, Y'')$	σ'
$\mathsf{SelectO}_{S1}(\epsilon, 100, \Sigma_O) = \{0, 1\}$	$\mathsf{SelectO}_{S2}(\epsilon, 100, Y') = \{0, 1\}$	$MinD(1, \{0, 1\}) = 1$	$(100, 1)$
$\mathsf{SelectO}_{S1}((100, 1), 100, \Sigma_O) = \{0, 1\}$	$\mathsf{SelectO}_{S2}((100, 1), 100, Y') = \{0, 1\}$	$MinD(1, \{0, 1\}) = 1$	$(100, 1) \cdot (100, 1)$
$\mathsf{SelectO}_{S1}((100, 1) \cdot (100, 1), 001, \Sigma_O) = \{0, 1\}$	$\mathsf{SelectO}_{S2}((100, 1) \cdot (100, 1), 001, Y') = \{0, 1\}$	$MinD(0, \{0, 1\}) = 0$	$(100, 1) \cdot (100, 1) \cdot (001, 0)$

Table 2.5: Serial composition scheme using Select()-output enforcement

event $x \in \Sigma_I$, the set of output events y in Y that allow to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q \setminus \{q_v\}$ with (x, y) is defined as:

$$\mathsf{SelectO}_{\mathcal{A}_\varphi}(q, x, Y) = \{y \in Y : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v\}.$$

For example, consider policy P illustrated in Figure 2.1a. We have $\mathsf{SelectO}_P(q_0, 01, \{0, 1\}) = \{0\}$.

$\mathsf{minD}(x, X')$ (resp. **$\mathsf{minD}(y, Y')$**): Consider X' (resp. Y') as a set of input (resp. output) events acceptable to all policies φ, and x (resp. y) as the original input (resp. output). $\mathsf{minD}(x, X')$ (resp. $\mathsf{minD}(y, Y')$) non-deterministically selects an edit $x' \in X'$ (resp. $y' \in Y'$) such that it is of minimum deviation from the original input event x (resp. output event y).

2.4.2 Incremental Enforcement Scheme using Select Functions

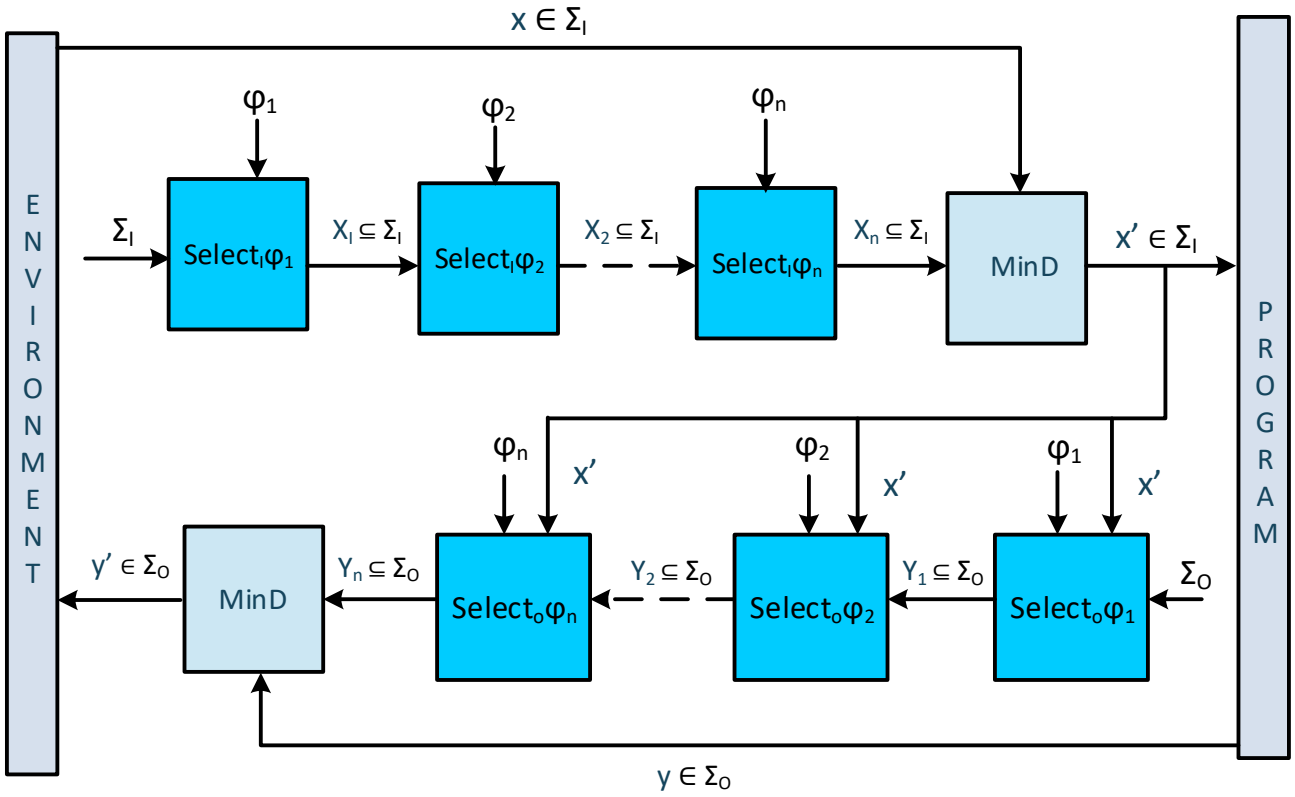


Figure 2.7: Incremental Enforcement via Serial Composition using Select functions

In serial composition using Select, the input enforcer E_I is implemented by the serial composition of SelectI() followed by MinD(). The input set Σ_I is fed to SelectI() in serial to pre-compute a valid input set satisfying all policies. As shown in Figure 8, $\mathsf{SelectI}_{\varphi_1}$ produces input set X_1 according to policy φ_{1I} from input Σ_I. Similarly, $\mathsf{SelectI}_{\varphi_2}$ outputs set X_2 w.r.t policy φ_{2I} from input X_1 and so on. The final input set obtained X_n satisfying all policies φ_1, $\varphi_2 \cdots \varphi_n$ is input to MinD(). Whenever an input x is received from the environment, it is fed to MinD() that selects a minimal edit $x' \in X_n$ such that $\sigma_I \cdot x'$ (σ_I already output) satisfies input policies.

Similarly, the serial composition of output enforcer E_O is implemented with the serial composition of SelectO() followed by MinD(). The output of input enforcer a' along with all possible output Σ_O are fed to SelectO() in serial to pre-compute a valid output set satisfying all policies. As shown in Figure 8, $\mathsf{SelectO}_{\varphi_1}$ chooses output set Y_1 according to policy φ_1 for input x'. Similarly, $\mathsf{SelectO}_{\varphi_2}$ chooses set Y_2 according to policy φ_2 from input Y_1 and so on. The final output set obtained Y_n satisfying all

policies φ_1, $\varphi_2 \cdots \varphi_n$ is input to MinD(). Whenever an output y is received from the program, it is fed to MinD() that selects a minimal edit $y' \in Y_n$ such that $\sigma \cdot (x', y')$ (σ_I already output) satisfies policies.

Definition 8 (Incremental enforcement via serial composition using select)**.** *Given two properties φ_1 and φ_2 (where φ_{1I} and φ_{2I} are their corresponding input policies), we define the enforcement function $E_{\varphi_1} \Rrightarrow E_{\varphi_2} : \Sigma^* \to \Sigma^*$ as $E_O(E_I(\sigma_I), \sigma_O)$ where :*

- *$E_I : \Sigma_I^* \to \Sigma_I^*$ is defined as:*

$$\begin{aligned} E_I(\epsilon_{\Sigma_I}) &= \epsilon_{\Sigma_I} \\ E_I(\sigma \cdot a) &= \sigma'_I \cdot MinD(a, \mathsf{SelectI}_{\varphi_2}(\sigma'_I, (\mathsf{SelectI}_{\varphi_1}(\sigma'_I, \Sigma_I)))) \end{aligned}$$

  where $\sigma'_I = E_I(\sigma)$.

- *$E_O : \Sigma_I^* \times \Sigma_O^* \to (\Sigma_I \times \Sigma_O)^*$ is defined as:*

$$\begin{aligned} E_O(\epsilon_{\Sigma_I}, \epsilon_{\Sigma_O}) &= \epsilon_\Sigma \\ E_O(\sigma_I \cdot x, \sigma_O \cdot y) &= \sigma' \cdot (x, y') \end{aligned}$$

  where $\sigma' = E_O(\sigma_I, \sigma_O)$

  $y' = MinD(y, \mathsf{SelectO}_{\varphi_2}(\sigma', x, \mathsf{SelectO}_{\varphi_1}(\sigma', x, \Sigma_O)))$.

Note that the serial composition of enforcers using Select functions always works. That is, given two policies φ_1, φ_2 and also $\varphi_1 \cap \varphi_2$ are all enforceable, serial composition of enforcers of φ_1 and φ_2 as per the above definition works and the final output obtained will always satisfy $\varphi_1 \cap \varphi_2$. Let us consider input enforcement to understand this (similar reasoning also applies to output enforcement). As per the serial composition Definition 8, all possible inputs Σ_I are fed to the input enforcers in serial composition, and the set obtained (using $\mathsf{SelectI}()$) is a valid one satisfying all input policies ($\varphi_1$and φ_2 in this case). When a new input word is received, it is input to the MinD() that chooses (if required) a suitable element from the valid set available with it.

Let us consider the following example to understand this further.

Example 7 (Serial composition scheme using Select()). *Let us again consider policies S_1 and S_2 illustrated in Figure 2.2. Both policies S_1 and S_2 are enforceable individually. The policy $S_1 \cap S_2$ is also enforceable. Also, when we compose input and output enforcers for these policies in series as per Definition 8, the final output obtained does satisfy policy $S_1 \cap S_2$. For example consider the word $(100, 1) \cdot (110, 1) \cdot (011, 0)$ to be processed incrementally is shown in Tables 2.4 and 2.5. Whenever any input is given, the MinD() always selects a valid element to input to the output enforcement function, always satisfying all the policies.*

Theorem 3 (Serial composition using select). *Consider two policies φ_1, φ_2 defined as SA, and where $\varphi = \varphi_1 \cap \varphi_2$. If policy φ is enforceable, then $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$ as per Definition 8 is an enforcer for φ (satisfies all the constraints as per Definition 4).*

The proof of Theorem 3 is given in Appendix **??**.The proof is based on induction on the length of the input word σ.

Remark 8. *Definition 8 supports incrementally adding a new enforcement layer. Whenever another new policy to be enforced φ_i is given, we can obtain the input and output select functions for the policy φ_i individually, and these can easily be plugged-in to enforce φ_i in addition to the previously enforced policies.*

Remark 9 (ORDER OF COMPOSITION OF ENFORCERS DOES NOT MATTER). *The order in which the input (resp. output) enforcers are composed, does not affect the final outcome in the proposed incremental enforcement approach. From the definition 8, we can see that each enforcer from the input set computes the set of all valid events w.r.t its policy, which is fed to the next enforcer in the sequence. The output set produced by the last enforcer in the sequence satisfies all the policies from which one element is chosen by MinD. Thus the order of input (resp. output) enforcers do not matter in the proposed incremental enforcement scheme.*

Chapter 3

Problem Statement

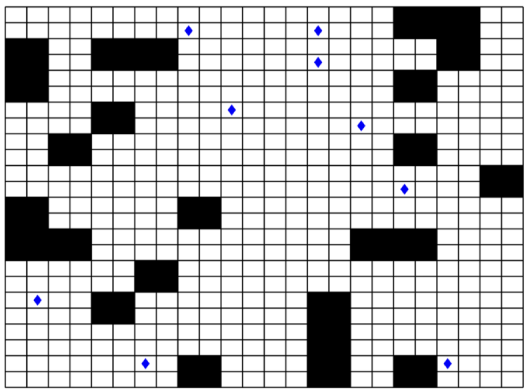
In this section, we investigate the application of compositional runtime enforcement in underwater robotic swarms and compare its effectiveness against the conventional monolithic approach. The focus is on enhancing a Multi-Robot Coverage Path Planning (MRCPP) algorithm tailored for mapping underwater vegetation, particularly within seagrass beds. This study evaluates how these runtime enforcement strategies influence key aspects of the MRCPP algorithm, such as performance, scalability, and adaptability, when operating in dynamic and unpredictable underwater environments. Through this comparison, we aim to identify a more efficient and robust framework for ensuring mission success in such challenging scenarios.

3.1 DARP: Divide Areas Algorithm for Optimal Multi-Robot Coverage Path Planning

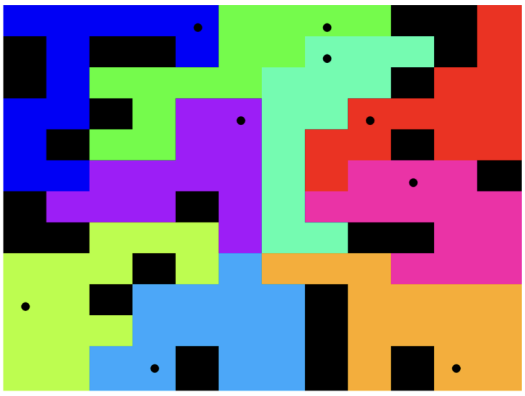
In this paper, we will be using the algorithm discussed in [26].

In essence, the DARP algorithm follows a cyclic coordinate descent optimization scheme updating each robot's territory separately but towards achieving the overall multi-robot Coverage Path Planning (mCPP) objectives. After the desired area divi-

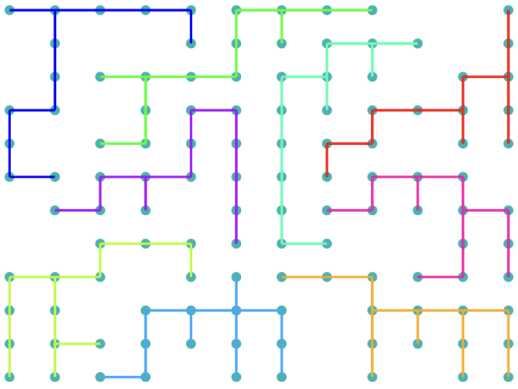
sion is achieved, we use the Spanning Tree Coverage (STC) algorithm to produce the optimal path for each robot, in order to achieve full coverage of the area of interest. This algorithm is able to account for prior-defined obstacles too.



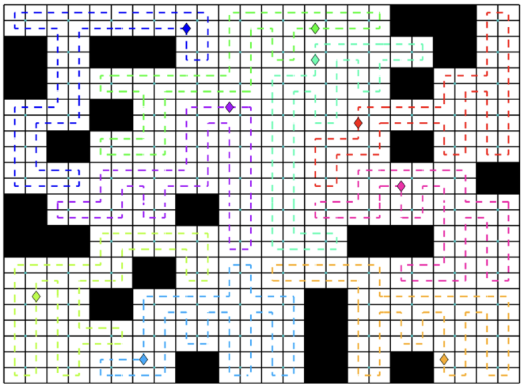
(a) Initial cells discretization, robots' cell, and obstacles



(b) DARP outcome - robots' exclusive areas



(c) Constructing Minimum Spanning Trees for each one of the robots sets



(d) Final Paths, designed to circumnavigate the MSTs

Figure 3.1: DARP+STC Proposed Approach, sample execution with 24x24 grid size, 9 robots and 100 obstacles [26]

There are n_r robots, with L_i being the robot sets for the i_{th} robot. A selection $L_1, L_2, ..., L_{n_r}$ poses an optimal solution for the mCPP, iff

- $L_i \cap L_j = \emptyset, \quad \forall i, j \in \{1, \ldots, n_r\}, \, i \neq j$
- $\bigcup_{i=1}^{n_r} L_i = L$
- $|L_1| \approx |L_2| \ldots |L_{n_r}|$
- L_i is connected, $\quad \forall i \in \{1, \ldots, n_r\}$
- $\chi_i(t_0) \in L_i$

The first condition ensures that each cell is assigned exclusively to one robot's set, enforcing a non-backtracking property in the resulting solution. The second condition specifies that the union of all L_i sets must cover every unblocked cell within the area, fulfilling the fundamental coverage requirement of completeness. The third condition focuses on maximizing the use of multi-robot dynamics by ensuring that the number of cells $|L_i|$ in each robot's set is approximately equal. The fourth condition mandates that each robot's set L_i should consist of compact and contiguous cells, forming a cohesive sub-region. This guarantees a fair division and enables seamless navigation within spatially connected areas, preventing robots from spending unnecessary time traveling between disjoint regions. Finally, the fifth condition requires that each robot's initial position $\chi_i(t_0)$ is located within its assigned set L_i, ensuring zero preparation time and minimizing energy consumption. Any algorithm capable of constructing the L_i sets while satisfying these five conditions can be combined with the STC algorithm to produce optimal solutions for the original mCPP problem.

Regarding the existence of such solutions, it has been proven that a fair partition, not restricted to convex regions, always exists for any polygon and any number of partitions. However, the variation of the problem considered here includes an additional condition: the requirement for each partition to contain a specified arbitrary point within the polygon. This added constraint makes the problem dependent on the arrangement of these arbitrary points and means a solution does not always exist. The overall formulation and proposed algorithm address cases where at least one optimal solution is guaranteed to exist.

In the algorithm, until all area division objectives are met, do:

1. Assign every (x, y) cell to a robot according to:

$$A(x, y) = \arg\min E_i(x, y)$$

   Here, E_i is the evaluation matrix for Robot i and $E_i(x, y)$ is the reachability from (x, y) and the initial position of robot i.

2. For each i-th robot, where $i \in \{1, \ldots, N\}$, perform the following steps:

   (a) Calculate k_i, where $k_i = |L_i|$

   (b) Update $m_i \leftarrow m_i + \eta(k_i - f)$, where m_i is a scalar correction factor for the i^{th} robot, and f denotes the global fair share.

   (c) Calculate the connectivity matrix C_i.

   (d) Update $E_i \leftarrow C_i \odot m_i E_i$.

After the desired area division is achieved, we use Spanning Tree Coverage (STC) algorithm to produce the optimal path for each robot, in order to achieve full coverage of the area of interest.

3.2 Drone Swarms

Underwater drones, also known as Unmanned Underwater Vehicles (UUVs), have found significant applications in both civilian and military domains. These versatile vehicles are used in tasks ranging from deep-sea exploration to military operations. A notable area of innovation is **multi-agent Coverage Path Planning (mCPP)**, which focuses on optimizing the coverage of a defined area using multiple coordinated UUVs. Below are key applications of UUVs leveraging mCPP, supported by real-world examples.

- **Military Applications:**
  - **Mine Countermeasures:** UUVs have been extensively used for mine detection and removal. For instance:
    - During the Iraq War, the US Navy deployed UUVs to clear mines around the port of Umm Qasr [27].
    - The REMUS UUV, a 3-foot-long robot, can clear mines in one square mile within 16 hours, a task that would take human divers over 21 days [28].

  - **Data Collection and Reconnaissance:** UUVs collect critical environmental data for military operations. Examples include:
    * The Chinese Sea Wing (Haiyi) UUV, found near Indonesia's Selayar Island in 2020, gathers data such as water temperature, salinity, and oxygen levels to map optimal submarine routes [29].
  - **Submarine Warfare and Surveillance:** Advanced UUVs perform key roles in submarine warfare. For example:
    * The Proteus UUV, developed by Huntington Ingalls Industries, utilizes synthetic aperture sonar to identify and neutralize underwater targets. This capability was demonstrated during the 2018 Advanced Naval Technology exercises [30].
  - **Multi-Domain Operations:** UUVs with trans-medium capabilities, such as China's flying submarine drones, extend mCPP applications to both underwater and aerial missions [31].

- **Deep-Sea Exploration and Research:**
  - **Mapping and Sample Collection:** UUVs map the ocean floor and collect samples efficiently. Examples include:
    * The *Sentry* UUV by the Woods Hole Oceanographic Institution, which can map the seabed at depths of up to 6,000 meters [32].
  - **Biodiversity Studies:** UUVs assist in exploring and monitoring underwater ecosystems:
    * Researchers have used UUVs to collect samples for studying the microplastic content of ocean floors and assess the health of deep-sea ecosystems [33].
  - **Exploration in Extreme Environments:** UUVs are critical for operations in high-pressure environments:
    * Bio-inspired soft robots capable of operating in the Mariana Trench have been used for deep-sea exploration and environmental monitoring [34].

- **Ecosystem Monitoring and Rehabilitation:**
  - **Water Quality Monitoring:** UUVs are used to monitor and manage water ecosystems. Examples include:
    - Duro AUS UUVs deployed in New York City's Randall's Island Park Alliance to monitor wetland health in the East and Harlem Rivers.
  - **River and Estuary Restoration:** UUVs support environmental rehabilitation initiatives:
    - The Bronx River Alliance uses UUVs to monitor and rejuvenate river ecosystems, helping to restore wildlife and improve water quality.

As advancements in robotics, artificial intelligence, and communication technologies continue to advance, the integration of mCPP in UUV operations is expected to expand, enabling transformative applications across military, scientific, and environmental domains.

This study investigates the application of compositional runtime enforcement in underwater robotic swarms, evaluating its performance in comparison to the conventional monolithic approach. The research focuses on integrating runtime enforcement into a Multi-Robot Coverage Path Planning (MRCPP) algorithm tailored for mapping underwater vegetation in seagrass beds. By analyzing both methods, the study seeks to determine their influence on the algorithm's efficiency, scalability, and flexibility in dynamic underwater conditions.

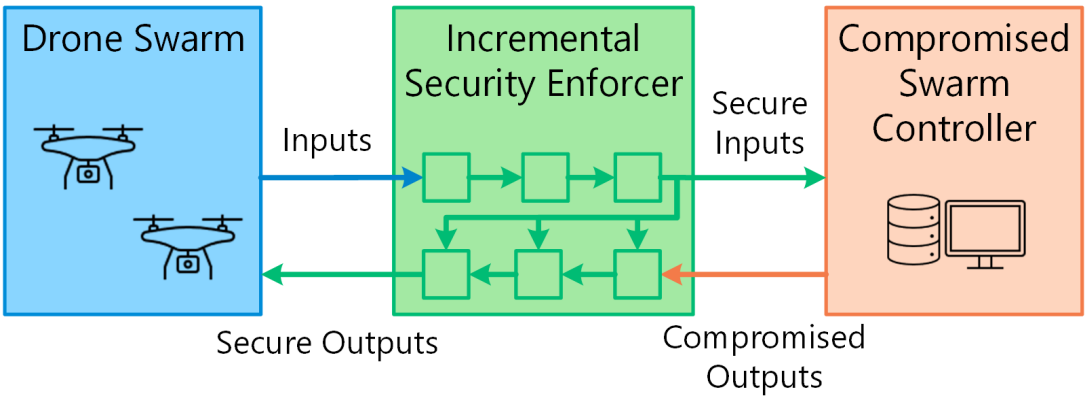


Figure 3.2: System diagram of incremental enforcement for a drone swarm.

3.3 Attack On Drones

While underwater drones offer significant benefits in exploration, research, and military applications, they are not immune to threats from malicious actors. Recent work by Yaacoub et al. in [8] surveys a range of attacks on robotic systems. Underwater drones are particularly vulnerable to interference and manipulation due to their reliance on acoustic communication and complex navigation systems. For instance, acoustic jamming and spoofing attacks can disrupt communication and navigation, leading to miscoordination or loss of control [35]. Additionally, acoustic or magnetic interference can trigger emergency behaviors, such as premature surfacing or drifting [36], and prevent the relay of vital status and control signals [37]. This disruption in communication between drones and the central controller can lead to unsafe outcomes, including collisions, mission failure, or environmental damage.

Hardware vulnerabilities, such as tampered communication modules or sensor systems [38], provide opportunities for attackers to inject, intercept, or alter control commands. Such attacks can force underwater drones to violate operational boundaries, collide with obstacles, or fail to execute their intended tasks.

In this work, we consider five potential attacks on an underwater drone swarm's coordinated operation:

A1 *Boundary Breach (Injection and Alteration)*: where the attacker manipulates control signals to force drone(s) to leave their designated operational area, potentially breaching restricted zones or damaging sensitive ecosystems.

A2 *Overwhelm Inputs (Injection and Alteration)*: an attack where drone(s) are overloaded with conflicting or erroneous control signals, leading to erratic behavior or mission disruption.

A3 *Block Control Signals (Jamming)*: any attack that disrupts acoustic or RF communication, preventing updated control signals from reaching one or more drone(s) in the swarm.

A4 *Drain Power (Injection and Alteration)*: where the attacker commands drones to perform energy-intensive tasks, such as high-speed maneuvers or prolonged thruster usage, depleting their batteries prematurely.

A5 *Pressure Limit Violation (Injection and Alteration)*: an attack where an adversary disrupts the simultaneous execution of pressure limit monitoring and ascent actions. This could cause drones to remain at unsafe depths, risking structural failure due to high pressure, or to ascend improperly, compromising mission integrity or safety.

We categorize these attacks into common types: *jamming* and *injection and alteration*. *Injection and alteration* attacks involve the addition or manipulation of control packets, while *jamming* attacks remove or block communication packets. To mitigate these threats, we develop robust policies enforced through a compositional runtime enforcement framework tailored to underwater drone operations.

3.4 Mitigation with Enforcement

This work defines a series of security policies to mitigate attacks on underwater drones. These policies are enforced through a runtime enforcement framework to ensure safe and secure operation, even in the presence of malicious interference. The inputs and outputs for each drone are defined as follows:

- Inputs (I): $\{\mathsf{left\_boundary}, \mathsf{right\_boundary}, \mathsf{down\_boundary}, \mathsf{up\_boundary}, \mathsf{rpm\_limit}, \mathsf{pressure\_limit}\}$ (signals from the plant to the controller).
- Outputs (O): $\{\mathsf{up}, \mathsf{down}, \mathsf{right}, \mathsf{left}, \mathsf{rpm\_up}, \mathsf{rpm\_down}, \mathsf{ascent}\}$ (control actions sent from the controller to the plant).

The following policies correspond to the identified attacks and define mitigation strategies:

- **Policy φ_1: Boundary Breach (Prevents A1)** This policy ensures that drones do not violate predefined operational boundaries. It enforces the following:
  - left_boundary and left cannot occur simultaneously.
  - right_boundary and right cannot occur simultaneously.
  - down_boundary and down cannot occur simultaneously.
  - up_boundary and up cannot occur simultaneously.
- **Policy φ_2: Conflicting Signals (Prevents A2)** This policy ensures no simultaneous conflicting control signals are sent to the drone. It enforces the following:
  - left and right cannot occur simultaneously.
  - up and down cannot occur simultaneously.
  - rpm_up and rpm_down cannot occur simultaneously.
- **Policy φ_3: Block Control Signals (Prevents A3)** This policy mandates that drones ascend to the surface when control signals are not received for a specific time threshold (5 seconds). It enforces:
  - If no input or output signals are received, and the timer exceeds 5 seconds, then the drone ascends to the surface.
- **Policy φ_4: Drain Batteries (Prevents A4)** This policy ensures that the rpm_up signal does not occur simultaneously with rpm_limit. It enforces:
  - rpm_up and rpm_limit cannot occur simultaneously.
- **Policy φ_5: Pressure Limit and Ascent Coordination (Prevents A5)** This policy ensures that pressure_limit and ascent occur simultaneously. It enforces:
  - pressure_limit and ascent must occur simultaneously.

These policies collectively address key threats to underwater drones and ensure secure and reliable operations in dynamic and potentially adversarial environments.

Policy φ_1 mitigating attack A1, *Boundary Breach*

The *Boundary Breach* attack occurs when a drone is instructed to violate predefined airspace boundaries or the separation limits between drones. To mitigate this, Policy φ_1 ensures that the drone operates within its allowed 2D workspace while moving in four primary directions: left, right, up, and down. The drone is prevented from breaching boundaries or failing to move when expected. Specifically, for the **left** direction:

- If the drone attempts to move left while already at the left boundary (`left_boundary` $\wedge$ `left`), it is considered a violation, and the drone's left movement is disabled (`left := 0`).
- If the drone fails to move left when not at the left boundary ($\neg$ `left_boundary` $\wedge \neg$ `left`), the policy forces left movement (`left := 1`).

The same logic applies for the other directions (**right**, **up**, and **down**) with their respective boundary conditions (`right_boundary`, `up_boundary`, `down_boundary`). This policy ensures that the drone adheres to boundary constraints, preventing it from breaching airspace limits or failing to navigate appropriately within the defined 2D area.

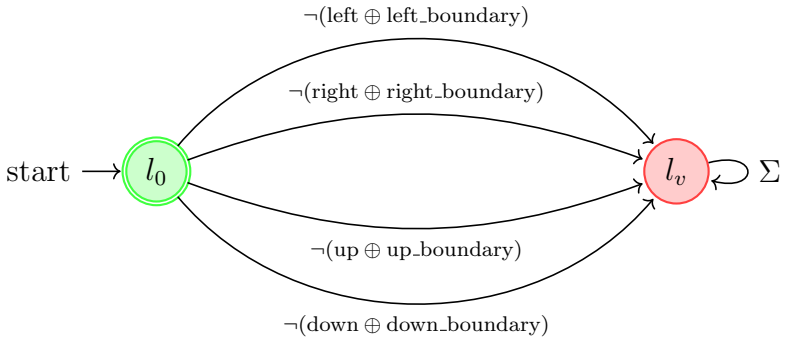


Figure 3.3: Automaton for policy φ_1 which prevents a drone from breaching its boundary

Policy φ_2 mitigating attack A2, *Overwhelm Inputs*

In attack A2, *Overwhelm Inputs*, the attacker sends conflicting control signals to the drone. Policy φ_2 is designed such that simultaneously present conflicting controls cause violation. The synthesised enforcer will therefore suppress one of these conflicting signals to ensure the drone is not overwhelmed.

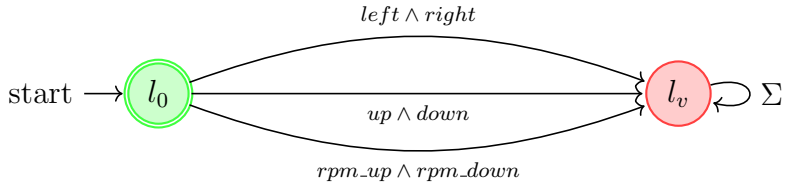


Figure 3.4: Automaton for policy φ_2 which prevents overwhelming of the inputs

Policy φ_3 mitigating attack A3, *Block Control Signals*

The attack A3, *Block Control Signals*, which jams signals prevents the drone from getting updated controls. This could result in it continuing to climb or move in undesirable ways. The policy detects if no control signals are sent in any 5-second period. The drone is then brought to the surface using the `ascent` signal.

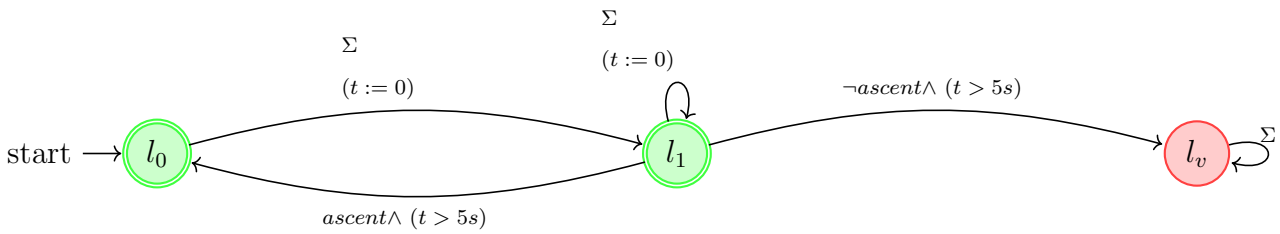


Figure 3.5: Automaton for policy φ_3 which ensures safe transitions based on timeout and ascent conditions.

Policy φ_4 mitigating attack A4, *Drain Batteries*

Attack A4, *Drain Batteries*, involves injection or alteration of signals to increase the drone's RPM beyond the efficiency threshold, which is crucial for maintaining optimal flight endurance during the QR display. Although higher RPM allows faster movement,

it significantly reduces efficiency, causing the battery to deplete more rapidly. To mitigate this, an RPM limit is enforced. Policy φ_4 is a simple policy which prevents *rpm_up* signal when *rpm_limit* is active.

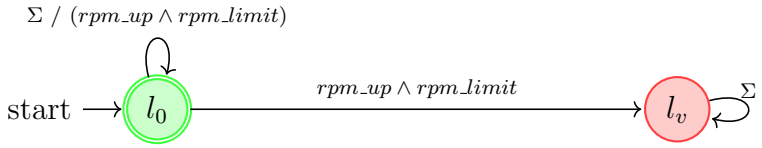


Figure 3.6: Automaton for policy φ_4 which ensures safe RPM limits to prevent battery drain.

Policy φ_5 mitigating attack A5, *Pressure Limit Violation*

Attack A5, *Pressure Limit Violation*, occurs when an adversary prevents the drone from initiating ascent even when the pressure limit is reached. This attack exploits the safety mechanism designed to protect the drone from operating at unsafe depths, potentially leading to structural damage or mission failure. To mitigate this, Policy φ_5 ensures that the drone immediately triggers the `ascent` signal whenever the `pressure_limit` condition is met, preventing prolonged exposure to high-pressure environments. This simple policy blocks operations that ignore the `pressure_limit` condition, ensuring safe and reliable behavior.

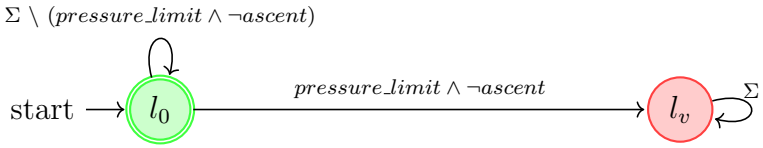


Figure 3.7: Automaton for policy φ_5 which ensures ascent is triggered when the pressure limit is reached.

Chapter 4

Implementation and Evaluation

To assess the effectiveness of the proposed incremental serial composition approach, we developed a straightforward simulator for a multi-drone system. The simulator is based on the structure depicted in Figure 3.2, incorporating a controller, enforcer, and the drone itself. This implementation enables a direct comparison between the traditional monolithic approach and the incremental serial composition method. For evaluation, we intentionally introduce faults into the simulated controller, compelling the enforcer to make consistent edits. This setup allows us to measure and analyze the time required for the enforcer to execute its corrections effectively.

The software tool *easy-rte* [39] was used to produce enforcers for this implementation. The tool provides support for monolithic composition, which we use to produce our monolithic enforcers. Additionally, *easy-rte-incremental* [23] was used to produce the serial implementation.

To ensure the enforcers actively intervene, the controller was intentionally compromised. To illustrate the effect of progressively complex compositions, we began with a single φ_1 policy, designed to counteract the boundary breach attack (A1) for the first drone. Additional policies, φ_2, φ_3, φ_4, and φ_5, were then incrementally incorporated, addressing attacks A2, A3, A4, and A5, respectively, across the drones.

Therefore we created five enforcers for both monolithic and serial compositions to satisfy the following:

$E_{\varphi 1}$: enforces φ_1

$E_{\varphi 2}$: enforces φ_1, and φ_2

$E_{\varphi 3}$: enforces φ_1, φ_2, and φ_3

$E_{\varphi 4}$: enforces φ_1, φ_2, φ_3, and φ_4

$E_{\varphi 5}$: enforces φ_1, φ_2, φ_3, φ_4, and φ_5

Monolithic compositions were composed into a single policy first and are denoted φ_{1M}, φ_{2M}, ..., φ_{55M}. These are then synthesized into enforcers denoted $E_{\varphi_{1M}}$. Serial compositions are denoted $E_{\varphi_{1S}}$, $E_{\varphi_{2S}}$, ..., $E_{\varphi_{5S}}$.

4.1 Evaluation Parameters

xmx

Appendix A

Title of Appendix

xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx
xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx
xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx
xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx xxxxxx.

$$y = \alpha\, x + n \tag{A.1}$$

Publications

Journal Publications

1.
2.
3.

Conference Publication

1.
2.

Bibliography

[1] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*. IEEE, 2008, pp. 363–369.

[2] G. Loukas, *Cyber-physical attacks: A growing invisible threat*. Butterworth-Heinemann, 2015.

[3] A. Humayed, J. Lin, F. Li, and B. Luo, "Cyber-physical systems security—a survey," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1802–1831, 2017.

[4] The Economic Times. (2024, November) Massive cyberattacks strike iran's nuclear facilities and government agencies: Is israel behind it? Accessed: 2024-11-26. [Online]. Available: https://economictimes.indiatimes.com/news/defence/massive-cyberattacks-strike-irans-nuclear-facilities-and-government-agencies-is-israel-behind-it/articleshow/114192558.cms?from=mdr

[5] R. Langner, "To kill a centrifuge: A technical analysis of what stuxnet's creators tried to achieve," *The Langner Group*, 2013.

[6] J. Slay and M. Miller, "Lessons learned from the maroochy water breach," in *International conference on critical infrastructure protection*. Springer, 2007, pp. 73–82.

[7] R. M. Lee, M. J. Assante, and T. Conway, "German steel mill cyber attack," *Industrial Control Systems*, vol. 30, no. 62, 2014.

[8] J.-P. Yaacoub, H. Noura, O. Salman, and A. Chehab, "Security analysis of drones systems: Attacks, limitations, and recommendations," *Internet of Things*, vol. 11, p. 100218, 2020.

[9] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.

[10] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier, "Runtime enforcement monitors: composition, synthesis, and enforcement abilities," *FMSD*, vol. 38, no. 3, pp. 223–262, 2011.

[11] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of nonsafety policies," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 3, pp. 19:1–19:41, Jan. 2009.

[12] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, A. Rollet, and O. Nguena Timo, "Runtime enforcement of timed properties revisited," *FMSD*, vol. 45, no. 3, pp. 381–422, 2014.

[13] M. Ngo, F. Massacci, D. Milushev, and F. Piessens, "Runtime enforcement of security policies on black box reactive programs," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 43–54.

[14] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of nonsafety policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 3, pp. 1–41, 2009.

[15] H. Pearce, M. M. Kuo, P. S. Roop, and S. Pinisetty, "Securing implantable medical devices with runtime enforcement hardware," in *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*, 2019, pp. 1–9.

[16] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[17] S. Adepu, S. Shrivastava, and A. Mathur, "Argus: An orthogonal defense framework to protect public infrastructure against cyber-physical attacks," *IEEE Internet Computing*, vol. 20, no. 5, pp. 38–45, 2016.

[18] S. Adepu and A. Mathur, "From design to invariants: Detecting attacks on cyber physical systems," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2017, pp. 533–540.

[19] A. Baird, H. Pearce, S. Pinisetty, and P. Roop, "Runtime interchange of enforcers for adaptive attacks: A security analysis framework for drones," in *2022 20th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 2022, pp. 1–11.

[20] Q. Sun, K. Zhang, and Y. Shi, "Resilient model predictive control of cyber–physical systems under dos attacks," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 7, pp. 4920–4927, 2019.

[21] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang, "Shield synthesis: Runtime enforcement for reactive systems," in *TACAS*, ser. LNCS, vol. 9035. Springer, 2015.

[22] S. Pinisetty, P. S. Roop, S. Smyth, S. Tripakis, and R. von Hanxleden, "Runtime enforcement of reactive systems using synchronous enforcers," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*. ACM, 2017, pp. 80–89.

[23] A. Panda, A. Baird, S. Pinisetty, and P. Roop, "Incremental security enforcement for cyber-physical systems," *IEEE Access*, vol. 11, pp. 18 475–18 498, 2023.

[24] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan 2003.

[25] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *Algebraic Methodology and Software Technology (AMAST'93)*. Springer, 1994, pp. 83–96.

[26] A. C. Kapoutsis, S. A. Chatzichristofis, and E. B. Kosmatopoulos, "Darp: Divide areas algorithm for optimal multi-robot coverage path planning," *Journal of Intelligent & Robotic Systems*, pp. 1–18.

[27] Editor-in-Chief, "The history of underwater drones," https://www.droneblog.com/the-history-of-underwater-drones/, September 2018, retrieved 2021-11-05.

[28] J. Carafano and A. Gudgel, "The pentagon's robots: Arming the future," Backgrounder 2093, 1-6. Available at: https://www.heritage.org/defense/report/the-pentagons-robots-arming-the-future, 2007, electronic version.

[29] The Guardian, "Indonesian fisher finds drone submarine on possible covert mission," https://www.theguardian.com/world/2020/dec/31/indonesian-fisher-finds-drone-submarine-on-possible-covert-mission, December 2020, retrieved 2021-11-05.

[30] Y. TADJDEH, "Annual naval exercise showcases unmanned underwater vehicle capabilities," *National Defense*, vol. 103, no. 780, pp. 24–26, 2018. [Online]. Available: https://www.jstor.org/stable/27022380

[31] T. Kadam, "China unleashes video of 'flying submarines'; beijing wants transmedia vessels to break enemy defenses," https://www.eurasiantimes.com/china-unleashes-video-of-flying-submarines-beijing-wants-cross-media/, November 2022, retrieved 2022-12-02.

[32] Woods Hole Oceanographic Institution, "Ships & technology used during the titanic expeditions," https://www.whoi.edu/know-your-ocean/ocean-topics/underwater-archaeology/rms-titanic/ships-technology-used-during-the-titanic-expeditions/, retrieved 2021-11-05.

[33] J. Barrett, Z. Chase, J. Zhang, M. Holl, K. Willis, A. Williams, B. Hardesty, and C. Wilcox, "Microplastic pollution in deep-sea sediments from the great australian bight," *Frontiers in Marine Science*, vol. 7, 10 2020.

[34] G. Li, X. Chen, F. Zhou, Y. Liang, Y. Xiao, C. Xunuo, Z. Zhang, M. Zhang, B. Wu, S. Yin, Y. Xu, H. Fan, Z. Chen, W. Song, W. Yang, B. Pan, J. Hou, W. Zou, S. He, and W. Yang, "Self-powered soft robot in the mariana trench," *Nature*, vol. 591, pp. 66–71, 03 2021.

[35] O. Westerlund and R. Asif, "Drone hacking with raspberry-pi 3 and wifi pineapple: Security and privacy threats for the internet-of-things," in *2019 1st International Conference on Unmanned Vehicle Systems-Oman (UVS)*. IEEE, 2019, pp. 1–10.

[36] A. H. Abunada, A. Y. Osman, A. Khandakar, M. E. H. Chowdhury, T. Khattab, and F. Touati, "Design and implementation of a rf based anti-drone system," *2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies, ICIoT 2020*, pp. 35–42, 2 2020.

[37] P. Valianti, S. Papaioannou, P. Kolios, and G. Ellinas, "Multi-agent coordinated close-in jamming for disabling a rogue drone," *IEEE Transactions on Mobile Computing*, 2021.

[38] A. Belous and V. Saladukha, "Hardware trojans in electronic devices," in *Viruses, Hardware and Software Trojans*. Springer, 2020, pp. 209–275.

[39] H. Pearce, S. Pinisetty, P. S. Roop, M. M. Y. Kuo, and A. Ukil, "Smart i/o modules for mitigating cyber-physical attacks on industrial control systems," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 7, pp. 4659–4669, 2020.