UQAC
Université du Québec
à Chicoutimi

A MULTI-TRACE MODEL FOR RUNTIME ENFORCEMENT AND VERIFICATION UNDER UNCERTAINTY

PAR

RANIA TALEB

THÈSE PRÉSENTÉE À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

DANS LE CADRE D'UN PROGRAMME EN EXTENSION DE

L'UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

EN VUE DE L'OBTENTION DU GRADE DE PHILOSOPHIÆ DOCTOR (PH. D.)

EN SCIENCES ET TECHNOLOGIES DE L'INFORMATION

QUÉBEC, CANADA

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

DEDICATION

In loving memory of my beloved grandfather, whose spirit continues to guide and inspire me, I dedicate this Ph.D. thesis.

Though you are no longer physically present, your words, etched in my memory, continuously uplift me, reminding me that setbacks are merely stepping stones on the path to success.

While you may not witness the culmination of this endeavor, I am certain that you are watching from above, your proud smile illuminating the heavens.

May your soul find eternal peace as I strive to make a difference in the world, carrying your legacy forward with every step I take.

In loving memory,

Your forever grateful granddaughter

ACKNOWLEDGEMENTS

I stand at the culmination of an extraordinary journey, and as I reflect upon the completion of this monumental undertaking, I am filled with a profound sense of gratitude. This Ph.D. thesis represents the collective efforts and support of numerous individuals who have left an indelible mark on my academic and personal growth.

First and foremost, I extend my deepest appreciation to my esteemed supervisors, Professor Sylvain Hallé and Professor Raphaël Khoury. Your unwavering guidance, intellectual rigor, and endless encouragement have been invaluable throughout this research endeavor. Your expertise and mentorship have shaped my scientific perspective and helped me navigate the intricate paths of academia. I am grateful for the countless hours you dedicated to reviewing my work, challenging my ideas, and pushing me to exceed my own expectations. My heartfelt appreciation also goes out to the faculty members and researchers in the Department of Computer Science and Mathematics / Université du Québec à Chicoutimi.

I extend my heartfelt thanks to the members of my thesis committee, Professor Djamal Robaine, Professor Marc Frappier, and Professor Yliès Falcone. Your insightful feedback, constructive criticism, and expert suggestions have significantly enriched the quality and depth of this thesis. I am indebted to each of you for your time, expertise, and commitment to fostering excellence in research.

I am grateful to my colleagues and friends who have provided a supportive and stimulating environment during my time as a Ph.D. student. Your camaraderie, shared experiences, and intellectual debates have made this journey all the more rewarding.

To my beloved family, thank you for your unyielding love, encouragement, and belief in my abilities. Your unwavering support and sacrifices have been the foundation upon which my academic pursuits were built. I am forever grateful for your understanding during late nights and missed family gatherings, knowing that your belief in my potential fueled my determination.

As I conclude this thesis, I humbly acknowledge the contributions of each and every person who has touched my life during this journey. Your collective support, guidance, and inspiration have shaped not only this thesis but also the person I have become. I am profoundly grateful for the privilege of standing on the shoulders of giants and for the opportunity to contribute to the advancement of knowledge in my field.

With deepest appreciation,

Rania Taleb

RÉSUMÉ

Au début des années 2000, on a constaté une augmentation significative de la complexité des systèmes logiciels, caractérisée par un nombre croissant de composants et d'interactions. Les méthodes traditionnelles de test et de vérification formelle se sont avérées insuffisantes pour garantir la précision et la sécurité de ces systèmes. De plus, il est devenu évident que des techniques d'analyse dynamique étaient nécessaires pour surveiller et analyser le comportement des logiciels en temps réel pendant leur exécution. Cela était crucial pour identifier et résoudre des actions imprévues ou malveillantes qui ne pouvaient pas être entièrement anticipées lors de la conception ou de l'analyse statique. En conséquence, le domaine de la vérification de l'exécution est apparu comme une discipline distincte en informatique. La vérification de l'exécution consiste à utiliser un système de surveillance pour observer le comportement d'un programme cible, en le traitant comme une séquence d'événements. L'objectif est de détecter les violations potentielles d'une politique de sécurité définie par l'utilisateur et de fournir un verdict vrai si la politique est respectée, ou un verdict faux si elle est enfreinte.

Le domaine de la vérification de l'exécution englobe un large éventail de sujets et fait face à divers défis. Un problème particulier concerne la nature des événements surveillés. Il n'est pas toujours réaliste de supposer que tous les événements sont complets, précis et arrivent dans l'ordre attendu. Plusieurs facteurs peuvent affecter les événements, entraînant une incomplétude, une imprécision ou des interruptions dans leur séquence d'arrivée vers le moniteur. En conséquence, le moniteur peut faire face à des difficultés à analyser avec précision les événements, ce qui conduit à des verdicts non concluants.

Pour résoudre ce problème, plusieurs solutions ont été proposées dans la littérature. Une approche consiste simplement à ignorer les événements imprécis et à les traiter comme s'ils ne s'étaient pas produits. Une autre solution consiste à construire des modèles probabilistes pour estimer la probabilité qu'un événement se soit produit et produire un verdict correspondant. De plus, une troisième solution consiste à remplacer les événements imprécis ou manquants par un ensemble de toutes les substitutions possibles, ce qui permet une analyse plus complète.

La troisième solution peut potentiellement générer un ensemble de verdicts au lieu d'un seul verdict. L'innovation clé de cette thèse réside dans le développement d'une méthode pour représenter les événements incertains et générer les substitutions possibles correspondantes. Cette innovation s'étend au langage de spécification utilisé pour définir la politique de sécurité et construire un moniteur capable de traiter les ensembles de toutes les substitutions possibles pour un événement tout en minimisant le temps d'exécution et les surcharges. Dans cette thèse, nous établissons un cadre logique qui utilise un proxy de contrôle d'accès étatique pour modéliser l'incertitude. Ce proxy a la capacité de transformer les événements en ensembles d'événements possibles, ce qui aboutit à ce que nous appelons une "multi-trace". De plus, nous présentons un algorithme pour transformer un moniteur traditionnel en un moniteur fiable et tolérant aux pertes. Le proxy et le moniteur sont tous deux des extensions de machines de Mealy. À travers des expériences menées dans différents scénarios, nous démontrons l'efficacité de notre approche dans la gestion de différents types de dégradation des données et de limitations d'accès.

La vérification de l'exécution est étroitement liée au concept de l'application à l'exécution (runtime enforcement), qui va encore plus loin en réagissant aux violations observées de manière à les corriger et à s'en remettre. Dans le processus d'application d'actions correctives à une séquence d'événements en entrée, plusieurs séquences correctes d'événements peuvent être générées. Cependant, la littérature existante ne propose pas suffisamment d'approches pour sélectionner la meilleure ou l'optimale séquence correcte. Cette thèse propose un pipeline d'application à l'exécution qui englobe l'altération de la séquence d'entrée par l'application des actions correctives nécessaires, garantissant la conformité à la politique de sécurité, et sélectionnant le remplacement optimal en fonction de critères spécifiques. Ces étapes sont divisées en trois modules distincts, offrant une approche modulaire qui simplifie le développement des moniteurs à l'exécution. Nous mettons en œuvre cette approche à l'aide du processeur de flux d'événements BeepBeep et démontrons son efficacité à travers des cas d'utilisation. L'évaluation expérimentale démontre que le cadre proposé permet de choisir dynamiquement des actions correctives appropriées à l'exécution, éliminant ainsi la nécessité de définir manuellement un moniteur d'application.

ABSTRACT

Runtime Verification is the process of observing a sequence of events produced by a running software system and determining whether this sequence complies with a specified property expressed using a formal notation. It is commonly believed that a monitor possesses full access to the event trace. However, there are numerous scenarios where the monitor functions with a certain degree of uncertainty regarding the trace's content. In this thesis, we define a logical framework where uncertainty is modeled by a stateful access control proxy that has the capacity to transform events into sets of possible events, resulting in what we refer to as a "multi-trace". We also provide an algorithm to lift a classical monitor into a sound, loss-tolerant monitor. Both the proxy and the monitor are extensions of Mealy machines. Experiments conducted on various scenarios demonstrate that our approach can effectively account for various types of data degradation and access limitations. Furthermore, our approach provides a tighter verdict than existing works in some cases and preserves the scalable performance of the model.

In other scenarios, it is crucial for the underlying system to adhere to specific security policies. In such cases, runtime enforcement can be employed to ensure the respect of a user-specified security policy by a program. This is achieved by providing a valid replacement for any misbehaving sequence of events that may occur during the program's execution. However, depending on the capabilities of the enforcement mechanism, multiple possible replacement sequences may be available, and the current literature lacks guidance on how to choose the optimal one. Additionally, the current design of runtime monitors imposes a substantial burden on the designer, as the monitoring task is typically accomplished by a monolithic construct, often an automata-based model. This thesis addresses these issues by proposing a new modular model of enforcement monitors, where the tasks of altering the execution, ensuring compliance with the security policy, and selecting the optimal replacement are split into three separate modules. This modular approach simplifies the creation of runtime monitors. We implement this approach using the event stream processor BeepBeep, and a use case is presented to demonstrate its effectiveness. Experimental evaluation shows that our proposed framework can dynamically select appropriate enforcement actions at runtime, eliminating the need for manual definition of an enforcement monitor.

INTRODUCTION

Runtime Verification (RV) or Runtime Monitoring has gained increasing interest in recent years [99]. It can be defined as the process of observing the behavior of a running system, determining whether the execution under study is compliant with the expected behavior of the system, and detecting any violations [124]. The running behavior is represented by the execution trace (the sequence of events produced by the system). The expected behavior is usually specified as a set of rules or formal properties that must be obeyed. A property generally involves conditions on the sequence of events, as well as the data inside these events.

In contrast to other formal verification methods, such as model checking [53] and theorem proving [35], which are typically performed offline, RV can be conducted online while the system is executing. Instead of relying on a model of the system and its environment, which can be extremely complicated and potentially result in a state explosion problem, RV works directly with the actual system. Furthermore, RV provides an advantage over exhaustive software testing [148] by analyzing a single execution trace at a time, rather than considering all possible input sequences and scenarios.

The process of collecting the trace of events and presenting it to the monitor is critical. Events can be collected from various sources, such as the events gathered from system instrumentation [17, 40, 96, 122, 187, 188] or external values measured and recorded by sensor devices [85]. Moreover, there is no general convention on what format the events should take in the trace. Many notations and formats can be used to represent events, depending on the monitoring framework employed [157].

Regardless of the variety of event sources, most RV approaches assume that the monitor has complete and error-free access to the trace of events against which to evaluate a given property [32, 97, 103]. However, there are multiple situations where this assumption does not hold, such as in the case of incorrect system instrumentation, imprecise measurements, sampling techniques applied in RV to control overhead, and misconfiguration of data access control policies, among others. In this respect, a recent Dagstuhl seminar report has emphasized the importance of dealing with incomplete, imprecise, and faulty sources of events [16], as did a recent survey of challenges related to RV [161]. Ignoring the fact that incomplete and imprecise events might have occurred gives poor monitoring results. A sound and complete monitor should have a reasonable level of certainty about the content of the underlying trace that allows it to produce a conclusive verdict.

A variety of works have tackled the problem of RV with incomplete, uncertain, or missing information in the past decade. However, these approaches vary greatly in several dimensions of the problem, which makes them difficult to compare. Some of these techniques sacrifice soundness and may produce imprecise verdicts. Other techniques depend on the recovery of lost events for a sound and meaningful verdict. Others try to come up with a conservative approximation of the possible verdict regardless of the unknown events in the gap, while others propose new specification languages or extend existing ones with operators that allow the monitoring of some properties in the presence of incomplete events and the production of sound verdicts in some situations. Even the reason why incomplete or uncertain data may occur or the way in which a "perfect" trace becomes incomplete varies from one work to the next.

In fact, the problem of RV under uncertainty is relatively recent. Each contribution presents its approach in isolation without discussing its relation to other similar works. Consequently, we face an extremely fragmented vision of the state of the art on this question, making it difficult to identify areas where further research is needed. In this thesis, we introduce our approach to RV of traces with partial information. We also describe and synthesize different approaches from the literature that aim to handle RV for systems with incomplete traces,

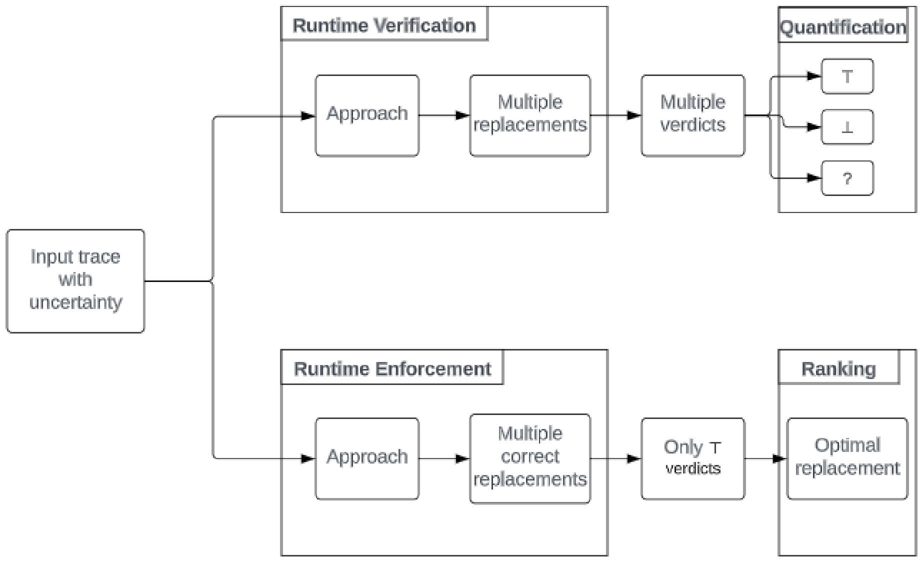


Figure 0.1 : Runtime Verification and Runtime Enforcement Frameworks.

employing formal, statistical, and other techniques.

When relying solely on RV, its capability is limited to determining whether a property is satisfied or not. In situations where incomplete events are present, various methods can be employed to obtain a certain verdict. One approach, similar to the one presented in this thesis, involves generating multiple potential replacements for a missing event, resulting in multiple traces and different verdicts (a positive "⊤" or negative "⊥" or inconclusive "?" verdict for each replacement). To assess the likelihood of each verdict's occurrence, quantification can be employed. However, in safety-critical systems where failures can have severe consequences, an additional layer of safety assurance becomes essential. To address this need, runtime enforcement serves as a valuable complement to RV. By actively monitoring the program's behavior, runtime enforcement can detect safety violations and intervene accordingly. This intervention plays a crucial role in preventing accidents, safeguarding human lives, and preserving physical assets. Acting as a security enforcement paradigm, runtime enforcement ensures compliance with user-defined security policies, as outlined in [75].

Runtime enforcement effectively replaces any identified misbehavior during program execution with valid alternatives, guaranteeing adherence to the specified security policy. Each of the resulting traces is a correct replacement of the input trace (only positive “$\top$” verdicts are produced). As incorrect traces can be altered in various ways, resulting in multiple potential replacements, a specific criterion can be used to rank each replacement. The output trace that best aligns with the specified policy can then be selected based on this ranking. Figure 0.1 depicts the two frameworks available for application to an input trace: RV and runtime enforcement.

Depending on the capabilities of the enforcement mechanism, multiple possible replacement sequences may be available, and the current literature is silent on the question of how to choose the optimal one. Furthermore, the existing design of runtime monitors places a significant burden on designers, as they are typically implemented as a monolithic construct, such as an automata-based model. In this thesis, we introduce a novel modular model for enforcement monitors that addresses these limitations. Our approach separates the tasks of modifying the execution, ensuring compliance with the security policy, and selecting the optimal replacement into three distinct modules, simplifying the creation of runtime monitors. We demonstrate the practical implementation of this framework using the event stream processor BeepBeep and present a use case to illustrate its effectiveness. Experimental evaluation confirms that our proposed framework can dynamically select suitable enforcement actions at runtime, eliminating the need for manual enforcement monitor definition.

The approaches and results presented in this thesis have been disseminated through the following publications:

- In the 9th IEEE/ACM International Conference on Formal Methods in Software

Engineering, FormaliSE@ICSE 2021, held in Madrid, Spain from May 17-21, 2021, our paper titled *"Runtime Verification Under Access Restrictions,"* [169] was featured. Notably, this publication was honored with the Best Paper Award, highlighting the profound impact and significance of our research in the field of runtime verification with partial information.

- Additionally, our work titled *"A Modular Runtime Rnforcement Model Using Multi-traces,"* [168] was presented at the Foundations and Practice of Security - 14th International Symposium, FPS 2021, which took place in Paris, France from December 7-10, 2021. Subsequently, this paper was expanded into a journal publication entitled *"A Modular Pipeline For Enforcement Of Security Properties At Runtime,"* [171] which is currently in press in the Annals of Telecommunications, expected to be released in 2023.

- Furthermore, we have contributed a paper titled *"Uncertainty In Runtime Verification: A Survey,"* to the Computer Science Review journal. Currently, this paper is under review and awaiting publication.

The remainder of the thesis is structured as follows:

1. Chapter 1 is an explanation of several formal notations and related concepts in RV and enforcement. The chapter details the mathematical underpinnings of traces, events, policies, truth domains, and other related concepts and the various ways a security policy can be specified.

2. Chapter 2 presents a comprehensive state-of-the-art analysis. It introduces the problem statement, discusses the different causes of partial information, and describes and compares various approaches in RV and enforcement.

3. Chapter 3 presents our abstract model for accounting for access restrictions in a monitoring context. It introduces the concept of an access proxy and its role in modeling events with access restrictions. The chapter also explores the process of lifting a loss-tolerant monitor from a classical monitor and quantifying the final verdicts.

4. Chapter 4 outlines our approach to runtime enforcement of a security policy. It describes our pipeline, which consists of different types of proxies for altering the trace. The chapter provides a formal definition and a thorough discussion of the two key concepts underlying our notion of enforcement: correcting the input sequence and selecting the optimal corrective course of action.

5. Chapter 5 showcases the implementation of the access control proxy and the enforcement pipeline as an extension of the BeepBeep pipeline. The chapter presents experiments conducted in various scenarios to measure the memory and time overhead of the loss-tolerant monitor. Additionally, experiments are performed to demonstrate the effectiveness of our access control proxy in accounting for certain types of data degradation, which can only be addressed in related works through an over-approximation of uncertainty. The chapter also empirically compares the impact of different enforcement strategies and scoring functions for the same policy and input sequence. Furthermore, it compares the overhead of enforcing the property using our pipeline with that of a conventional automaton model.

6. Chapter 6 serves as the concluding chapter of the thesis.

CHAPTER I
RV

RV serves as a valuable complement to offline verification techniques like model checking and theorem proving, as well as partial solutions like testing and debugging [99]. By combining the exhaustive nature of offline verification methods with the ability to apply them to actual program traces, as seen in testing and debugging, RV offers the best of both worlds. However, this benefit comes at a cost. The primary challenge in integrating RV into a system is managing the resulting runtime overhead, which can arise from various factors, such as monitor invocation, computation and evaluation of property predicates based on the program's state, potential performance slowdown due to program instrumentation and trace extraction, and potential interference between the program and the monitor, as they may share resources.

Another challenge in RV is the source and type of events available to the monitor. An event is not necessarily an observation detected during system execution. It can refer to a wide variety of phenomena outside the system, such as events recorded by environmental sensors that capture data from the system's surrounding environment (temperature, humidity, pressure, or light). These events can take on numerical forms, such as integers or decimals. External devices, such as cameras and microphones can also capture events in the form of images and audio clips. Messages transferred through a network, such as HTTP requests, can also be considered events of the text type (strings).

A specific RV problem is defined by the format used to represent events in the traces produced by a system, as well as the specification language that represents conditions (referred to as "properties") over these events. To some extent, these different combinations have been shown to be translatable into each other [142], although there may be some loss of expressiveness when the formats differ. This chapter begins by establishing the formal notations and fundamental concepts that will be employed throughout the entirety of this thesis. Additionally, we provide an elucidation of the concepts of runtime verification and runtime enforcement.

1.1 FORMAL NOTATIONS AND BASIC CONCEPTS

In this section, we provide formal definitions and examples for one of the main components of any runtime verification or enforcement framework is the *security policy*, the task of which is to define what is a "correct" input.

Another important component in this thesis is the *transducer*, which has the power of transforming an input into a modified output. The transducer will be comprehensively defined as part of our RV model in Section 3.1 and enforcement model in Section 4.1, where we will provide detailed explanations and descriptions. In the present context, inputs and outputs will be taken as sequences of arbitrary data objects called *events*. To this end, let Σ be a finite or countably infinite set of elements called *events*. For simplicity, we focus on atomic events, but the framework presented in this thesis is easily generalized to parameters of data-bearing events. The set of all finite sequences from Σ, also called *traces*, is given as Σ^*. Given a trace $\overline{\sigma} \in \Sigma^*$, we use the notation $\overline{\sigma}[i]$ to range over the elements of $\overline{\sigma}$, where i represents the event at the i-th position (the first event is at $i = 0$). The notation $\overline{\sigma}[i..]$ denotes the remainder of the sequence starting from action $\overline{\sigma}[i]$, while $\overline{\sigma}[..i]$ denotes the prefix of $\overline{\sigma}$, up to its i-th position.

The concatenation of two sequences $\overline{\sigma}$ and $\overline{\sigma}'$ is given as $\overline{\sigma} \cdot \overline{\sigma}'$. The empty sequence is denoted ϵ, and $\overline{\sigma} \cdot \epsilon = \epsilon \cdot \overline{\sigma} = \overline{\sigma}$. As usual, the notation $\overline{\sigma}' \preceq \overline{\sigma}$ denotes that $\overline{\sigma}'$ is a prefix of $\overline{\sigma}$. Given a sequence $\overline{\sigma}$ and a set $S \subset \Sigma^*$, we override notation by letting $\overline{\sigma} \cdot S$ denote the set: $\bigcup_{\overline{\sigma}' \in S}\{\overline{\sigma} \cdot \overline{\sigma}'\}$. In the same way, if S and S' are two sets of traces, $S \cdot S'$ is defined as

$\bigcup_{\overline{\sigma} \in S} \overline{\sigma} \cdot S'$.

1.1.1 SECURITY POLICIES

Based on the above elementary notation, we can now provide a formal definition of security policies. Although we use the term "security" to describe these policies, they can be more broadly interpreted as the definition of what constitutes a valid input. This notion can vary depending on the context; for example, a policy could represent the expected ordering of operations defined by some network protocol, and valid inputs according to this policy would correspond to protocol-compliant sequences.

1.1.1.1 DEFINITION

A security policy is a subset $\Phi \subseteq \Sigma^*$ of sequences called the *valid* sequences. For example, given an abstract alphabet $\Sigma = \{a, b\}$ made of only two events, the policy stated informally as "a must be the first event" corresponds to the set $\{a, aa, ab, aaa, aab, aba, abb, \dots\}$ made of all finite traces that start with a. Typically, a policy circumscribes an infinite subset of Σ^*, although this is not a requirement.

This set Φ induces a function $\varphi : \Sigma^* \to \mathbb{B}_4$, which associates to every trace a value called the *verdict*. The set $\mathbb{B}_4 = \{\top, \top^?, \bot_?, \bot\}$ corresponds to four possible "Boolean" outcomes, with $\top$ and $\bot$ respectively meaning "true" and "false". The remaining two values, which can intuitively be interpreted as "possibly true" and "possibly false", represent a form of uncertainty in the verdict. A verdict that is either true or possibly true will be called a *positive* verdict; similarly, a verdict that is either false or possibly false will be called a *negative* verdict. A verdict that belongs to $\{\top, \bot\}$ is said to be *definitive*, otherwise, it is called *uncertain*. Formally, function φ is defined as follows:

$$\varphi(\overline{\sigma}) = \begin{cases} \top & \text{if } \overline{\sigma} \in \Phi \land \forall \overline{\sigma}' \in \Sigma^*, \overline{\sigma} \cdot \overline{\sigma}' \in \Phi \\ \top^? & \text{if } \overline{\sigma} \in \Phi \land \exists \overline{\sigma}' \in \Sigma^* \text{ s.t. } \overline{\sigma} \cdot \overline{\sigma}' \notin \Phi \\ \bot_? & \text{if } \overline{\sigma} \notin \Phi \land \exists \overline{\sigma}' \in \Sigma^* \text{ s.t. } \overline{\sigma} \cdot \overline{\sigma}' \in \Phi \\ \bot & \text{if } \overline{\sigma} \notin \Phi \land \forall \overline{\sigma}' \in \Sigma^*, \overline{\sigma} \cdot \overline{\sigma}' \notin \Phi \end{cases}$$

When $\varphi(\overline{\sigma})$ returns true ($\top$), it indicates that the policy is currently satisfied and will remain so forever, regardless of events that can be appended to it. For example, a simple policy stating that "event a eventually occurs" becomes true for a trace as soon as it contains a. In the same way, when $\varphi(\overline{\sigma})$ returns false ($\bot$), it indicates that the policy is currently violated and is irremediably so. The policy stating that "event a should never occur" becomes false for a trace as soon as it contains a, and whatever events are appended at the end of $\overline{\sigma}$ cannot change this fact.

As one may guess, the definition of the remaining two possible verdicts suggests that the fate of the security policy depends on what may come after. Verdict "possibly true" ($\top^?$) indicates that $\overline{\sigma}$ currently satisfies the policy but that there exists a continuation of that trace that does not belong to the security policy. For instance, the policy stating that "a must not occur" is satisfied by the trace consisting of the single event b, but there exists an extension of that trace that does not belong to Φ (namely the trace ba). Conversely, the policy stating that "a must eventually occur" is not satisfied by the trace consisting of the single event b, but there exists an extension of this trace that does belong to Φ (again, the trace ba). Consequently, this trace would be associated to the "possibly false" ($\bot_?$) verdict. One can then define the Boolean connectives on these four values by assuming a total order on $\mathbb{B}_4$ such that $\bot < \bot_? < \top^? < \top$. Then for $x, y \in \mathbb{B}$, we have that $x \land y \triangleq \min(x, y)$, $x \lor y \triangleq \max(x, y)$. Negation is defined as usual for $\top$ and $\bot$, and further $\neg\bot_? = \top^?$.

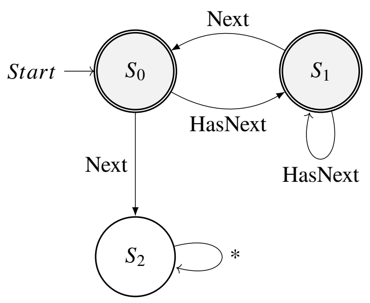


Figure 1.1 : A graphical representation of a finite-state automaton representing the constraint that $Next$ cannot be called before calling $HasNext$ first [123].

1.1.1.2 EXAMPLES OF POLICIES

In this section, we examine various security examples from existing literature and clarify how each policy can be precisely specified using formal specification languages. It is important to note that in Section 1.4, we thoroughly define and describe the majority of specification languages found in the existing literature.

Next and HasNext Pattern Policy: The standard Java API defines many interfaces in which the flow of methods invoked on objects must follow specific patterns to be utilized correctly [13, 41]. Such patterns are described in the documentation using several rules where the violation of these rules can cause the program to misbehave or throw an exception. A common example of this situation concerns the methods HasNext and Next of the iterator interface. The proper use of an iterator stipulates that one should never call method next() before first calling method HasNext(). The correct ordering of these calls can be expressed by a finite-state machine shown in Figure 1.1 and can also be described by an LTL formula as follows:

$$\mathbf{G}(\mathbf{X}\ \text{Next} \rightarrow \text{HasNext})$$

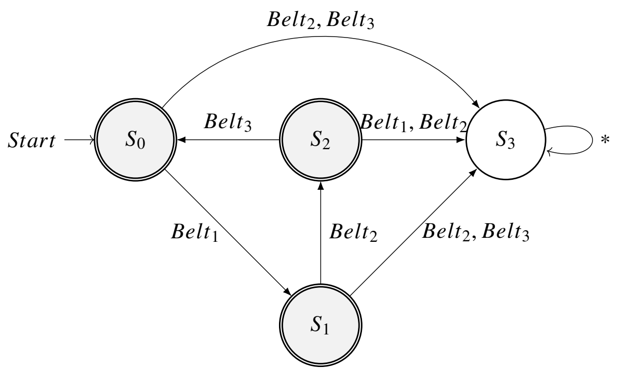


Figure 1.2 : Parcel Dispatcher Policy Automaton [70].

Fair Parcel Dispatcher Policy: In their work, Falcone *et al.* [70] considered a dispatcher who can accept parcels and distribute them to three conveyor belts. The initial behavior of the dispatcher is arbitrary in the sense that the dispatcher can move a parcel to any belt. The desired property states that *"the dispatcher is fair in the sense that it distributes the parcels on the belts one after the other in a specified order"*. The specification is modelled using the automaton given in Figure 1.2, which defines the fair re-partition among three conveyor belts following the order $Belt_1$, then $Belt_2$, then $Belt_3$. The alphabet of the property is $\Sigma = \{Belt_1, Belt_2, Belt_3\}$. The accepting states are S_0, S_1 and S_2.

Privacy Policies in Online Social Networks: On online social networks (OSNs) [145], there are many contexts and time-dependent dynamic policies that can be expressed using static operators as well as represented using a deterministic automaton with transitions labelled by events that the online social network can perform. For instance, the policy *"Co-workers cannot see my posts while I am not at work, and only family can see my location while I am at home"* can be expressed using the static policy operator $\mathcal{F}_g(x)$ to denote that anyone in group g is forbidden from performing action x, where x can refer to the forbidden action such as posting, seeing a location, etc.) and $\mathcal{F}_{\overline{g}}(x)$ to denote the complement of a group of users g. Then the policy, while not being at work, can be expressed as $\mathcal{F}_{co-workers}(read\text{-}post)$,

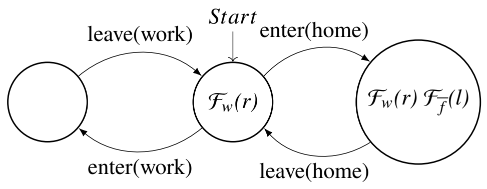


Figure 1.3 : A finite-state machine for the OSN policy [145].

and the policy when not at home to be $\mathcal{F}_{\overline{Family}}$*(see-location)*. By synchronizing with the actions of the social network application registering the arriving and leaving actions of the user (enter(l) and leave(l) respectively), the policy can be represented by a finite-state machine as in Figure 1.3. For simplicity, we use $\mathcal{F}_w(r)$ and $\mathcal{F}_{\overline{f}}(l)$ to represent $\mathcal{F}_{co-workers}$*(read-post)* and $\mathcal{F}_{\overline{Family}}$*(see-location)*, respectively.

File Format Policy: An example from [151] considers a scenario where an application writes a non-empty sequence of characters from the set {a, b, c} to a file, through multiple write operations, and a policy specifying that *"At the end of the sequence of writes, the file's content must respect a specific format where each string should end with a special character {!, ?}, which cannot occur elsewhere in the string"*. Hence, the input alphabet is Σ = {a,b,c,!,?}. The property can be specified by an LTL formula as follows:

$$(a \vee b \vee c)\, \mathbf{U}\, ((! \vee ?) \rightarrow \mathbf{XG} \perp)$$

where $\mathbf{G}\perp$ indicates that no more input actions can be accepted. The policy can also be specified using a finite-state machine as shown in Figure 1.4. The initial state is S_0 and the only accepting state is S_3.

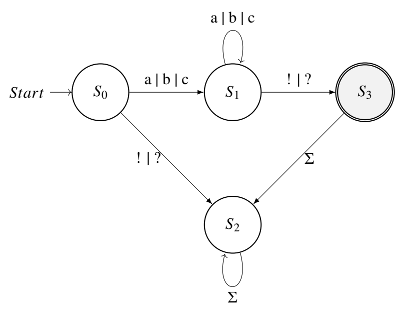


Figure 1.4 : A finite-state machine representing the property that "*each string should end by a special character* ! *or* ?" [151].

1.1.2 MONITORABILITY

A crucial question to consider is whether a property expressed using a specific formalism can be effectively monitored, and if the monitoring process can yield a definitive verdict [25].

In the context of RV, given a property φ and an execution trace, the goal is to examine a prefix $\overline{\sigma}[..i]$ of the trace and determine whether it belongs to the set $[[\varphi]]$ (i.e., $\overline{\sigma}[..i] \in [[\varphi]]$) or not (i.e., $\overline{\sigma}[..i] \notin [[\varphi]]$), where $[[\varphi]]$ represents the collection of all traces whose prefixes satisfy the property φ, i.e. the language $L(\varphi)$.

The RV monitor is restricted to observing a finite sequence of events within the trace and reaching a verdict based on that sequence. Once a verdict ($\top$ or $\bot$) is issued, the monitor is expected to maintain its stance without being influenced by subsequent events. In other words, the verdict should remain consistent for any prefix extension, even when there are infinitely many potential extensions. However, in certain cases, it may be necessary to extend the finite sequence to assess the monitorability of φ over infinite executions and ultimately arrive at a conclusive verdict. Nevertheless, there are properties for which it is impossible to determine a definitive and fixed verdict, regardless of any further extensions.

Consider the following two properties:

- P_1: G_p states that p always holds.

- P_2: $G(p \rightarrow F_q)$ states that whenever p holds, q holds in a future position.

For property P_1, once the monitor detects a prefix where p does not hold, it becomes evident that the property is not satisfied, and there is no need to extend it to an infinite execution. Therefore, it is possible to determine that the final verdict of P_1 over an infinite execution is $\perp$ by monitoring a finite sequence. On the other hand, for the second property P_2, even if the monitor receives a sequence where p holds at a state followed by a certain number of states where q does not hold, it remains uncertain whether this sequence can be extended to a correct execution until a state where q holds is reached. In this case, the monitor will be unable to reach a final verdict and will continue to produce inconclusive verdicts.

A property is considered monitorable if the monitor can consistently provide a conclusive verdict for every possible infinite extension of any trace. The issue of monitorability typically arises when attempting to relate finite sequences to infinite ones. If all infinite extensions of a given prefix $\overline{\sigma}[..i]$ belong to (or do not belong to) the language $[[\varphi]]$, we say that φ is monitorable (or not monitorable).

Monitorable properties can be classified into different classes. One classification scheme, known as the *safety-progress classification* [47], categorizes properties based on their behavior over infinite execution sequences. This classification includes two fundamental classes: *Class of Safety Properties* and *Class of Co-Safety Properties*.

Before delving into the details of these two classes, it is important to define the concepts of *good* and *bad* prefixes [31]. A bad prefix refers to a finite prefix that cannot be part of any accepting trace, while a good prefix is a finite prefix for which any infinite extension of the trace will be accepted. It is this classification that forms the foundation of the 3-valued semantics, where “bad prefixes” are mapped to *false*, “good prefixes” evaluate to *true*, and the remaining prefixes yield an inconclusive “?” result. Thus, monitors for 3-valued formulas classify prefixes as either good = true, bad = false, or “?” (neither good nor bad).

The *Class of Safety Properties* states that a property φ is a safety property if $\forall$ word $\sigma \in [[\varphi]]$, every prefix $\overline{\sigma}[..i]$ of σ is a good prefix. On the other hand, $\forall$ word $\sigma \notin [[\varphi]]$, $\exists$ at least one bad prefix violating φ. For example, the property $\varphi = G_p$ is a safety property because the property states that the atomic proposition p should hold in any state, so all prefixes of a word $\sigma \in [[G_p]]$ are good prefixes. Whereas, the property $\varphi = F_p$ is not a safety property because the word $\sigma = qqqpq$ satisfies F_p ($\sigma \in [[F_p]]$), however the prefix $\overline{\sigma}[..i] = qqq$ is a bad prefix of σ.

The *Class of Co-Safety Properties*, also known as *Guarantee Properties*, defines a property φ a co-safety property if, $\forall\sigma \in [[\varphi]]$, $\exists$ at least one good prefix satisfying φ. Conversely, $\forall\sigma \notin [[\varphi]]$, every prefix $\overline{\sigma}[..i]$ of σ is considered as a bad prefix. For example, consider the property $\varphi = F_p$, which is a co-safety property. It asserts that the atomic proposition p should eventually hold in the sequence. Therefore, $\forall\sigma \in [[F_p]]$, $\exists$ a good prefix $\overline{\sigma}[..i]$. On the other hand, the property $\varphi = G_p$ is not a co-safety property. There exist the word $w = pppqpp$ where p doesn’t hold ($\sigma \notin [[G_p]]$),although it does have a good prefix $\overline{\sigma}[..i] = ppp$.

An additional class is the *Obligation Class*, which can be obtained by combining finite conjunctions and disjunctions of safety and co-safety properties.

Classifying properties into specific classes simplifies the process of determining the monitorability of a given property by identifying the corresponding class. Various definitions of monitorability have been proposed based on the safety-progress classification.

The definition introduced by Kim *et al.* [116] focuses on monitoring to detect "Bad" behavior, which refers to violations of a property. This definition is based on safety properties, where any safety property is considered monitorable. The violation of a safety property in an infinite system execution can be identified by observing a finite sequence containing a bad prefix. For instance, the property G_p serves as an example

The definition of Pnueli and Zaks [153] extended the concept of monitorability and demonstratedd that a monitor can be employed to detect both "Good" and "Bad" behavior. It is essential to engage in monitoring only when there is a possibility of arriving at a conclusive verdict. Their definition of monitorability stated that a property φ is *positively* determined by a trace σ if all infinite continuations of σ satisfy φ. In such cases, the monitor assigns a verdict $\top$ to σ. Conversely, a property φ is *negatively* determined by a trace σ if all infinite continuations of σ violate φ. In such instances, the monitor assigns a verdict $\perp$ to σ. When the monitor is unable to reach a definitive verdict, it emits the non-conclusive verdict "?". Therefore, a property is considered monitorable over a trace if there $\exists$ a continuation of the trace where the property is either positively or negatively determined.

Based on the above definitions of monitorability, Bauer *et al.* [31, 32] demonstrated that both co-safety and safety properties can be monitored. Furthermore, they proved that the set of monitorable properties is a strictly larger set than the union of safety and co-safety properties. Additionally, it is possible for the set of monitorable properties to be a (strict) superset of the set of obligation properties [71, 73].

1.1.3 SOUNDNESS, COMPLETENESS AND MONOTONICITY

Soundness, completeness, and monotonicity are crucial concepts used to assess the effectiveness of RV approaches. Soundness measures the level of confidence one can place in the monitor's output. It determines the monitor's ability to provide an accurate verdict, indicating whether the monitored system has satisfied or violated the specified property, with no ambiguity or uncertainty.

Completeness, on the other hand, measures the level of confidence one can have in the monitor's ability to produce an output [74]. It refers to the extent to which an RV approach can capture all relevant events or behaviors of a system during runtime, ensuring that no violations of the expected properties are missed.

Monotonicity ensures that the verdicts obtained do not change when new events become available. In other words, if an RV approach reaches a conclusion about the system's behavior based on a set of observations, and subsequently more observations are made, the original conclusion should remain valid and not be invalidated.

1.2 WHAT IS RV?

A typical RV setup, as depicted in Figure 1.5, involves the creation of a monitor from a specification, the extraction of a trace from the execution of the target system, and the evaluation of this trace against the specified property. The specification property defines the desired behavior of the system and outlines what the system should or should not do in response to different inputs and varying circumstances. Typically, the property consists of a set of rules or constraints that must hold true for the system to fulfill its intended purpose. These properties can be expressed in various formal languages, such as temporal logic or automata.

Synthesizing an RV monitor from a property Depending on the specification language used, a property may not provide a direct algorithm for its evaluation on a trace of events. This is the case, for instance, with Linear Temporal Logic (LTL) [152], an extension of propositional

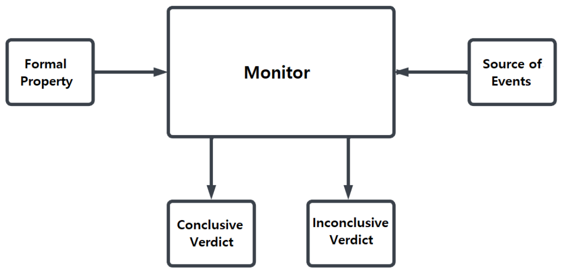


Figure 1.5 : RV Setup.

logic that allows assertions about the order of events in a sequence. Therefore, when applying run-time verification to a property expressed in a formal notation, it is often necessary to first create a *monitor* capable of concretely evaluating the property.

In the case of LTL, Bauer et al. propose a step-by-step method that takes an LTL property φ as input and produces a deterministic finite state machine (FSM) as output [32]. Figure 1.6 illustrates the steps involved. The initial step involves converting the LTL formula into a Non-deterministic Büchi Automaton (NBA) using one of several possible algorithms [19, 39, 66, 78, 84, 155, 180, 181]. An NBA is a type of automaton that accepts infinite sequences of states and can represent all possible executions satisfying an LTL property. The subsequent step involves simplifying the NBA by eliminating redundant states and transitions, a process achieved through algorithms such as the subset construction or the power set construction. The third step entails converting the reduced NBA into a Non-deterministic Finite Automaton (NFA), which accepts a finite sequence of states. This is accomplished by removing the acceptance condition from the NBA and transforming it into a transition system. The fourth step involves converting the NFA into a Deterministic Finite Automaton (DFA), which is an automaton with a unique transition for each input symbol and state. Finally, the DFA is mapped to an FSM by assigning state variables to each state and defining the transition function that determines the next state based on the current state and input variables.

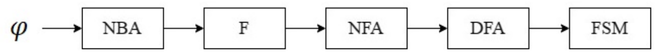


Figure 1.6 : Steps required to generate an FSM from an LTL formula φ.

System Instrumentation The system instrumentation step is a crucial stage in RV during which the monitor connects with the system that generates the events to be observed and processed [25]. In the case of a software system, instrumentation can be performed at the source code level [122], by adding additional code instructions to the source files before compilation. This allows tracking the execution of specific software components and generating an execution trace that can be used by the monitor. A similar approach can also be applied to compiled code at the binary level [131].

However, while early RV works mainly focused on instrumented software systems, the scope of potential event sources has expanded over the years. For instance, *system logs* can provide valuable insights into system behavior, including errors, warnings, and other events occurring during execution. Collecting data on various system parameters like CPU usage, memory usage, and disk I/O allows analysis to identify potential issues or areas for improvement.

Furthermore, one can consider systems with event sources from even more diverse origins. In fact, any event or data point relevant to system behavior can be instrumented and monitored for analysis, depending on the specific requirements and goals of the RV framework. For example, in software systems that interact with users, monitoring and logging user interactions offer valuable insights into user-system interactions, enabling the identification of potential issues and areas for improvement. In distributed systems or networked applications, monitoring network traffic provides insights into system behavior and helps identify performance, security, or communication-related issues among system components. In systems interacting with physical sensors, monitoring and analyzing sensor data offer insights into system behavior, aiding in identifying issues related to sensor accuracy, calibration, or data processing.

Analyzing System Execution Following the instrumentation phase, the retrieved events are transmitted to the monitor for analysis. This process is commonly referred to as execution analysis. The monitor examines the trace one event at a time. Monitoring can occur either *offline*, where the execution trace is stored in a log and provided to the monitor, or *online*, where event analysis is performed in real-time during execution in a synchronized manner [74].

The monitor interacts with the system by emitting a *verdict* for each consumed event, indicating the status of the property at that point in the execution. In its simplest form, the verdict domain can be represented as $\mathbb{B}_2 = \{\bot, \top\}$, where $\top$ represents a true verdict indicating that the property is satisfied, and $\bot$ represents a negative verdict indicating that the property is violated. However, most RV systems aim to provide more nuanced results and utilize verdict domains with three or more values. A common domain is $\mathbb{B}_3 = \{\bot, ?, \top\}$, where "?" denotes insufficient information to conclude either satisfaction or violation, indicating that the monitor cannot produce a definitive verdict in the current system state. Finer-grained verdicts with four or even five truth values have also been explored [31].

The monitor can also communicate with the system by providing *feedback*, allowing appropriate corrective actions to be taken if a property is violated. This aspect forms a separate field of study called *Runtime Enforcement* [69, 129, 163], which extends the realm of RV by aiming to modify the trace itself. This is achieved through event deletion, insertion, or modification to *correct* any illicit behavior present in the trace, rather than solely detecting it. As such, the monitor acts as a transducer, replacing the original, potentially invalid execution with an alternative, updated execution that demonstrably adheres to the desired property.

The aforementioned stages of RV have been applied in numerous diverse scenarios. These include monitoring programs to verify if their execution satisfies a given property [44, 59], monitoring and recovery of web service applications [89, 147, 166] where web services or other web-based implementations serve as event sources, monitoring of driving emissions in vehicles [117], bug detection in video games [182], verification of the behavior of aerial drones [140], and various other applications.

1.3 EVENTS AND EVENT TYPES

In the context of runtime verification, an event refers to a significant occurrence or observation that takes place during the execution of a system or program. Events play a fundamental role in runtime verification as they serve as the basis for analysis and decision-making by the monitoring system. By examining events and their ordering or properties, the monitor can evaluate the system's compliance with specified properties, detect violations, and provide verdicts or feedback accordingly.

The nature and interpretation of events depend on the specific context and requirements of the runtime verification task at hand. For example, values read and recorded by sensor devices, regardless of their type (strings, numbers, etc.), can be considered as events. Similarly, inner actions performed in a software system, such as returning search results, adding users to a database, or reading/writing to a file, can also be treated as events. Additionally, snapshots of the system's status taken at regular intervals can be categorized as events.

A *trace*, on the other hand, represents the linear sequence of events measured or produced by the execution of the system. There are various formats and notations available to represent events, depending on the specific requirements and conventions of the monitoring approach being used [157].

```
event, map, collection, iterator
updateMAp, 6750210, ,
createColl, 6750210, 2081191879,
createIter, , 2081191879, 910091170
useIter, , , 910091170
updateMap, 1183888521, ,
```

Figure 1.7 : An example of a trace of CSV events.

In this section, we will enumerate the most common types of events and demonstrate how each event type can be represented.

Events as Atomic Symbols In the simplest case, an event is a name for something that can happen, such as *openFile* or *closeFile*, or it can carry a value, such as a string, number, or a Boolean from the domain $\mathbb{B}_2 = \top, \bot$. Although this is the simplest and least structured form of an event, several works in the literature have used this notation [105, 106, 167].

CSV Events While atomic events are suitable in some situations, in many cases, events need to be represented in a more structured form. One possibility is to represent an event as a *tuple* composed of attributes and values, similar to data in a CSV (Comma-Separated Values) file. In fact, CSV events can be seen as tuples where each line of the file represents an event, and each element within the line corresponds to the value of an attribute for that event. An example of a CSV trace is shown in Figure 1.7. In this example, the first "line" provides the names of four attributes, and the subsequent lines represent individual events, with each value corresponding to its respective attribute. It is important to note that this notation allows events to have empty values for certain attributes.

Tuple-based events are commonly used in runtime verification. For instance, Havelund

```
<sensor-data>
  <reading timestamp="2023-02-27T10:30:00" type="temperature">
    <value unit="Celsius">25</value>
  </reading>
</sensor-data>
```

Figure 1.8 : A sensor event represented in XML.

et al. presented a benchmark for evaluating RV tools where traces of events are represented in CSV format [98]. Similarly, during the latest RV competition, CSV files were utilized to track Java operations on maps. Previous works have even suggested translating the task of RV on tuple events into the evaluation of an equivalent database query [179]. The CSV or tuple format is also employed in various other RV approaches, such as the approach by Ayesha et al. [109], Jonas et al. [149], and Vikas et al. [15].

XML and JSON Events XML, which stands for eXtensible Markup Language [132], is commonly associated with web services [88]. Data is expressed in XML using a tree structure. One common way to structure events in XML is by defining a root element that contains one or more child elements, each representing a specific event. Each event element can have attributes that describe the event, such as a timestamp, event type, or other relevant metadata. The content of each event element can include additional data associated with the event, such as event parameters or payload data.

For example, let's consider a simple XML representation of a sensor reading event as depicted in Figure 1.8. In this illustration, the root element is the sensor-data element, which contains a single reading element representing a sensor reading event. The reading element has two attributes (timestamp and type) that provide metadata about the event. The value element holds the actual sensor reading value (25) and an attribute (unit) specifying the unit of measurement. The "tags" are the syntactical feature employed to represent elements in a text

```
{
  "sensor-data": {
    "reading": {
      "@timestamp": "2023-02-27T10:30:00",
      "@type": "temperature",
      "value": {
        "@unit": "Celsius",
        "#text": 25
      }
    }
  }
}
```

Figure 1.9 : A sensor event represented in JSON.

file.

One of the primary advantages of using XML (Extensible Markup Language) is its wide support and standardization, allowing it to be parsed by numerous existing libraries in various programming languages. The XES format is an IEEE initiative aimed at standardizing the representation of event data in XML [3]. Many RV frameworks utilize XML-based formats for representing events, such as the LogFire framework [100], the JRec runtime monitoring framework for web services [22], the AXML runtime monitoring framework for XML documents [20], and the XMonitor runtime monitoring framework [55].

JavaScript Object Notation (JSON) is also employed for representing structured data. Instead of using “tags”, JSON employs a simpler syntax consisting of key-value pairs, arrays, and nested objects to represent an event.

The aforementioned example can be represented in JSON as demonstrated in Figure 1.9. By convention, the “@” symbol is used to denote attributes, and the “#” symbol is used to represent the text content.

Some RV frameworks utilize JSON to represent events, including FLINT [23], Umbral [119], Varan [141], Panda [101], and Medusa [7].

Events as Predicates An even more flexible approach to representing events involves modeling them as a set of *predicates* [27]. Formally, given a set of objects S, a predicate can be defined as a function $p : S^n \to \mathbb{B}_2$, where n is referred to as the *arity* of the predicate. With a fixed set of predicates $p_1, \ldots, p_m$ (each potentially having a different arity), an event can be represented as a function that specifies the value of each predicate for every possible argument.

To illustrate, let's consider a simple scenario where the set of objects consists of two light bulbs $S = a, b$, and the predicate $on : S \to \mathbb{B}_2$ represents the state of a light bulb (on or off). In this context, a possible event could be $on(a) = \top, on(b) = \bot$, indicating that light bulb a is on and light bulb b is off. A trace then becomes a sequence of such events, where the definition of each predicate may change from one event to the next, reflecting the varying data content.

This basic model can be extended to allow predicates with multiple arguments and predicates where each argument can be drawn from a different set. It can be observed that this representation encompasses (i.e., is more general than) the previous formats, as tuples or nested structures can be represented using a set of appropriately defined predicates.

Snapshots Thus far, the considered event types consist of individual data units representing a single "state" or "action." However, events can consolidate multiple such states or actions into a single data structure, potentially losing information about their actual content and ordering in the process. These are referred to as snapshots of events [184]. Figure 1.10 illustrates a snapshot of two data variables recorded by a *life data recorder* (LDR), a device that captures

	0	1	2	3
x	2	3	2	1
y^1	-	3	2	-
y^2	-	2	1	3
y^3	-	4	3	-
y^4	4	-	2	4

Figure 1.10 : An event of type snapshot.

updates to a set of variables generated by a medical device.

The snapshot in the figure consists of four "frames" that record variations in the values of two variables: x (occurring once per frame) and y (occurring at most four times per frame, with a dash entry indicating no recorded value for y). Representing these values as a snapshot is motivated by the lack of knowledge about the exact order of variable x in relation to the multiple variations of variable y. As a result, each recorded frame serves as an abstract representation of several event traces, where each event is a tuple (x, y). Formally, one possible trace of variable updates between frame 0 and frame 1 in Figure 1.10 can be represented as:

$$(2,4) \xrightarrow{x} (3,4) \xrightarrow{y} (3,3) \xrightarrow{y} (3,2) \xrightarrow{y} (3,4)$$

if the value of x changes before any change of y. Another trace can be:

$$(2,4) \xrightarrow{y} (2,3) \xrightarrow{x} (3,3) \xrightarrow{y} (3,2) \xrightarrow{y} (3,4)$$

if the value of x changes between the first and second change of y.

1.4 SPECIFICATION LANGUAGES

The definitions in Section 1.1 are agnostic to the formalism used to represent the security policy Φ. This feature has several advantages, notably flexibility and the possibility to integrate it with a variety of existing security mechanisms. However, there exist classical ways of representing a security property. Each property is an expression represented using one of several specification languages. In the following, we describe some of the them presented in the literature.

Regular Expression [25] A regular expression is a popular declarative language for describing sets of strings. As the correct execution of a system often relates to the possible ordering of observed events, a natural way to express properties is by considering them as patterns that must be matched against a sequence of symbols.

A regex comprises a sequence of characters describing a search pattern in a text. A typical regex mixes raw symbols with special characters that can be used to represent multiple alternatives or a form of repetition. For instance, a period (".") matches any character, while a range ("[∼]") matches any of the characters contained within the brackets. In addition, quantifier characters can be affixed to a symbol to indicate that the match may occur a variable number of times. Thus, "x?" indicates that x can be observed zero or one time, while "x+" indicates that x may be present at least once. Finally, alternation characters such as "|" represent the logical OR operator, so that "$x \sim | \sim y$" indicates that either x or y must be observed.

Regexes can be used to describe a regular language pattern and express a property. As an example, consider the policy stating that *a red light should be immediately followed by a green light*. The language of this pattern is a collection of strings over the alphabet $\Sigma = \{green, yellow, red\}$. Using regular expression operators, this can be expressed as follows:

$$(\text{green} \mid \text{yellow})^* \text{ red green}^+ (\text{green} \mid \text{yellow})^*$$

Some RV monitors accept regular expressions as their specifications, such as JavaMOP [48] and SEQ.OPEN [83].

Finite-State Automata [6] An automata is a computational model used to describe the behavior of a system that can exist in a finite number of states and transition between those states in response to input. Formally, it can be defined as a quadruple $M = \langle \Sigma, S, s_0, \delta, S_F \rangle$, where Σ is the set of input characters, S is a set of states, $s_0 \in S$ is the initial state, $\delta : S \times \Sigma \longrightarrow S$ is the transition function, and $S_F \subseteq S$ is the set of final or accepting states. With each input, the automaton moves from the current state to the next state using the transition function and eventually ends up in one of the final states. If the transition function allows at most one next state for any given state and input symbol, the automaton is called a Deterministic Finite Automaton (DFA) [165]. If the transition function is replaced by a transition relation, the automaton is referred to as non-deterministic.

Figure 1.11 represents the traffic light property using a finite state automaton. Here, s_0 is the initial state and both s_1 and s_2 are final states. The automaton transitions from s_0 to s_1 when encountering a red light event (r) where it should check whether the next input event is g or not. If g appears, the automaton returns to s_0, else if a non-green event appears, then the automaton moves to the final state s_2 and is stuck there producing the same output until the end of the input trace.

An expansion of NFA is the Probabilistic Automaton (PA) [154], which incorporates the likelihood of a particular transition into the transition function, resulting in a transition

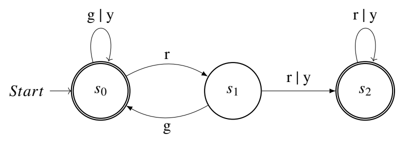


Figure 1.11 : A finite-state automaton representing the traffic lights property.

matrix. The class of languages recognized by probabilistic automata is referred to as stochastic languages, which includes regular languages as a subset. The number of stochastic languages is incalculable. In contrast to DFA and NFA, a PA employs a weighted set or vector of next states. These weights must total 1, representing probabilities, which makes it a stochastic vector.

LTL (Linear Temporal Logic) [152] An alternative way of specifying conditions on sequences of events is to turn to logic-based notations. LTL is built up from a finite set of propositional variables AP, over which expressions can be constructed using logical operators ($\neg$, $\vee$, and $\wedge$) and temporal modal operators (**G**, **F**, **X**, and **U**). These operators are called "future time" as they express conditions that hold from some starting point in a sequence and for subsequent events.

If φ represents a condition, the expression $\mathbf{G}\varphi$, for example, stands for "globally" and means that the formula φ must hold *globally*, i.e., for every suffix of the current trace. On the other hand, $\mathbf{F}\varphi$ stands for *eventually* and stipulates that φ should hold at some point in the future. The expression $\mathbf{X}\varphi$ stands for "next," meaning that φ should hold in the suffix of the trace starting from the *next* event. Finally, the binary operator **U** stands for "until"; the expression $\varphi_1\mathbf{U}\varphi_2$ means that φ_1 has to hold at least until φ_2 becomes true, and φ_2 must hold at some point in the future. Several operators can be combined to represent complex conditions on the accepted ordering of events in a trace, such as $\mathbf{G}\,\neg a \wedge \mathbf{F}\,b$, stating that a should never hold, and b must finally hold. The traffic light property can be expressed in LTL as follows:

$\mathbf{G}\,(red \rightarrow (\mathbf{X}\,green))$.

Several efforts have been made to augment LTL with quantitative operators that can represent quantitative (metric) real-time properties that are beyond the scope of classical LTL. Given the plethora of these logics, we will only emphasize the significant ones. For a more comprehensive explanation, readers can refer to the cited sources. Metric Temporal Logic (MTL) [118] is the most extensively scrutinized and investigated real-time extension of LTL. As MTL holds significant prominence among other extensions, it will be explained in greater detail in the next section.

Past-LTL [152] extends LTL with temporal operators that refer to past events, allowing the expression of properties such as "a has always been true in the past." Interval LTL [11] also extends LTL with operators that allow the specification of properties over intervals of time, such as "a holds for at least k time units within every interval of length n." Probabilistic LTL [21] where probabilistic operators are introduced into LTL, allowing expressing properties with a degree of uncertainty, such as "with probability p, eventually a happens." Quantified LTL [45] extends LTL with quantifiers, enabling the expression of properties over subsets of the state space, such as"for all states satisfying condition C, A holds eventually." Finally, LTL-FO$^+$ extends LTL with quantifiers on data values inside events [89].

TK-LTL: Tally Keeping-LTL [111] This specification language extends the semantics of LTL with several syntactic structures aimed at providing a quantitative evaluation of a different aspect of the trace. We briefly recall the semantics of its important operators; the reader is referred to [111] for complete details. One feature of TK-LTL is the use of a counter $\widehat{C}^{v}_{\varphi}$ ranging over the execution under consideration, where φ is an LTL formula and v ranges over the truth values of LTL, returns the number of suffixes of the input trace for which the evaluation of φ evaluates to v. We write $\widehat{C}^{\geq ?}_{\varphi}$ as a stand-in for $\widehat{C}^{?}_{\varphi} + \widehat{C}^{\top}_{\varphi}$ and $\widehat{C}^{\leq ?}_{\varphi}$ for $\widehat{C}^{?}_{\varphi} + \widehat{C}^{\perp}_{\varphi}$. The values returned by counters range over $\mathbb{N}$, but arithmetic operators or functions can be freely applied to the outputs of multiple counters over the same sequence to compute information about the trace, yielding a value in $\mathbb{R}$.

Aside from $\widehat{C}$, TK-LTL defines other counters: the unary $\mathcal{D}_{\varphi}^{v}$ counter returns the initial point in the input trace where a given property holds, the binary counter ${}_{\phi}\mathcal{D}_{\varphi}^{v}$ the first position at which a condition holds, starting from the satisfaction of another condition while the counter $\mathcal{L}_{\varphi}^{v}$ returns the index of the last occurrence of an event for which the property φ evaluates to v. The semantics of these counters, along with usage examples, are given in [111].

In addition to counters, the semantics of TK-LTL include quantifiers that examine the value returned by a counter for each prefix of the input sequence and return a value from the same three-valued truth domain as an LTL-property according to a condition sub-scripted to the quantifier. As is common in RV, quantifiers in LTL range over sets of executions. TK-LTL defines three quantifiers: the existential and the universal quantifiers with natural semantics, such as the formula $\exists_{=5}\widehat{C}_{p}^{\top}$ which returns $\top$ if the atomic proposition p holds on at least five prefixes of the input trace, and returns "?" otherwise; and the formula $\exists_{<0}\widehat{C}_{p}^{\top} - \widehat{C}_{q}^{\top}$ returns $\top$ if there exists a prefix of the input trace for which the atomic proposition q holds more often than p. The third quantifier is the propositional quantifier and is written as $\mathcal{P}$. The formula $\mathcal{P}_{\sim k}\widehat{C}$ thus evaluates to $\top$ if the comparison $n \sim k$ holds where n is the value returned by $\widehat{C}$. For example, let $\sigma = aaaba$ be a trace; the formula $\mathcal{P}_{=3}\widehat{C}_{a}^{\top}$ evaluates to $\top$ at positions $i = 3$ and $i = 4$, and to $\perp$ elsewhere.

Quantifiers, such as $\forall_{\sim k}\widehat{C}$ or $\exists_{\sim k}\widehat{C}$, verify whether the values returned by a counter meet a given condition c and return a verdict in V. This allows unlimited recursion of alternating LTL formulae, counters and quantifiers.

MTL: Metric Temporal Logic [118, 120] LTL can be extended with timing constraints using the MTL propositional bounded-operator logic. Temporal operators (such as 'until', 'next', and 'since') are augmented with time references. The **U** operator of LTL is replaced with $\mathbf{U}_I$, where I is an interval of reals with endpoints in $\mathbb{N} \cup \{\infty\}$. MTL can express deadline properties, meaning that the system is required to react within a specified time frame after a particular action takes place. For example, consider the property that *"every alarm is followed by a shutdown event in 10 seconds unless all clear is sounded first"*. This can be expressed in MTL as: $\Box(alarm \to (\Diamond_{(0,10)} allClear \vee \Diamond_{\{10\}} shutdown))$, where $\Box$ means *always*, $\Diamond$ means *eventually*, $(0, 10)$ means *'within 10 seconds'* and $\{10\}$ means *'in exactly 10 seconds'*.

MTL can be applied to linearly ordered time domains, which may be represented as discrete, dense, or continuous. The interpretation of MTL varies depending on the selected time flow, and its semantics may change accordingly. For example, suppose that $f : \mathbb{R}^+ \to 2^\Sigma$ is a mapping from a real-time point $t \in \mathbb{R}^+$ to the set of propositions holding at time t. Semantically, in a dense time, we have that $f \models \varphi_1 \cup_I \varphi_2$ if $\exists t \in I$ such that $f^t \models \varphi_2$ and $\forall t' \in (0, t) : f^{t'} \models \varphi_1$, where $f^t(s) = f(t + s)$. MTL can also represent the trace as a sequence of timed words. A time word σ is a finite or infinite word $(t_0, a_0)(t_1, a_1) \ldots \in (\mathbb{R}^+ \times \Sigma)$, where the sequence of t_i is strictly monotonic and non-zero. The semantics in this case can be as follows: $\sigma[i] \models \varphi_1 \cup_I \varphi_2$ iff $\exists j \geq i$ such that $\sigma[j] \models \varphi_2, (t_j - t_i) \in I$, and $(\forall i \leq k < j)\ \sigma[k] \models \varphi_1$.

LOLA: Logic Of Linear Arithmetic [60] LOLA is a temporal logic-based language that allows users to specify temporal properties over streams using various logical operators, such as conjunction, disjunction, implication, and negation. A stream of events is the same as the trace of events used in other languages; however, a stream can be thought of as an infinite sequence of real data values continuously generated and consumed.

LOLA accepts a specification in the form of a set of stream equations using typed stream variables. The output streams are computed from a given set of input streams. It has been shown that the expressiveness of LOLA exceeds that of FSM, LTL, and MTL (described in Section 1.4) because it can handle quantitative constraints over real-valued variables. A stream can be computed using values from other streams by using arithmetic operators, logical operators (such as $\wedge$, $\vee$, etc.), temporal operators (such as $Until$, etc.), and other operators to combine streams. It also allows a stream to be defined by referring to the value of an event in another stream k positions behind, using the construct $s[-k, x]$. If $-k$ corresponds to an offset beyond the start of the trace, the value x is used instead. For example, the stream $s_1 = t_1[+1, \text{false}]$ is obtained by taking, at each position i, the value corresponding to another stream t_1 at position $i + 1$, except at the last position, which assumes the default value false. Moreover, the language provides the expression $\text{ite}(b; s_1; s_2)$, which represents an if-then-else construct: the value returned depends on whether the predicate of the first operand evaluates to true.

LOLA can be used to model RV as a stream computation. Consider the specification property *"every red light should not be followed by a yellow light"*; suppose that g , r, and y are two input streams of Boolean events, representing green, red, and yellow light events, respectively. Using LOLA, the property could be expressed as follows:

$$t := y[1, false]$$
$$\varphi := \neg(r \wedge t)$$

The equation t checks if the next event is yellow, except at the last position, which assumes the default value false. The equation φ returns False whenever $\neg(r \wedge \neg t)$ is True, i.e., whenever a red light appears and a yellow light appears in the next position (t evaluates to

False), and True otherwise. This output can be used as the monitor verdict for the property.

As seen, the stream runtime verification (SRV) as pioneered by LOLA is specialized for specifying synchronous streams, which means that events arrive in discrete steps where every input stream has an event at every step, and all output streams produce an event. This is suitable for monitoring correctness properties and performing quantitative measures. However, it is not appropriate for processing events that arrive at different frequencies and have arbitrary real-time timestamps, such as in cyber-physical systems, where timing is a critical issue.

TeSSLa: Temporal Stream-Based Specification Language [57] TeSSLa is an asynchronous specification language that natively supports timestamped events. It mandates a global order for all stream events, but it doesn't necessitate all streams to have events occurring simultaneously. This enables modeling high-frequency streams.

An event stream in TeSSLa can be specified over a time domain $\mathbb{T}$ and a data domain $\mathbb{D}$ as a finite or infinite sequence $s = a_0 t_0, a_1 t_1 \in \mathbb{TD}$. To model a specification property, the language has many well-defined operators that can be used to transform an input stream of events into another stream. Given an input stream $write$ that provides write events to a file, it can be represented as $write = wt_0, -t_1, wt_2, wt_3, -t_4, -t_5, wt_6 ...$, where w denotes a write event and $-$ denotes no event. The following specification checks whether the time lapse between two write events exceeds 5 time units.

$$difference := time(write) - last(time(write), write)$$

$$output := filter(difference > 5, difference - 5))$$

The $time(write)$ operator accesses the timestamp of each event in the $write$ stream. The $last$ operator applies the $time(write)$ operator to the previous event. The stream $difference$

computes the time difference between the current w event and the previous one. The stream $\mathit{difference}-5$ is filtered by the condition $\mathit{difference} > 5$ using the $filter$ operator. The resulting stream $output$ represents a sequence of output verdicts.

Note that TeSSLa is enriched with many other operators, such as the $delay$ operator, which can create events at certain points. For example, the above property can raise a unit event on the $output$ stream as soon as we know that there was no write event:

$$timeout := const(5)(write)$$

$$output := delay(timeout, write)$$

The first equation maps the values of events to the constant value of 5, which is then used as the timeout value. In other words, the $timeout$ stream is derived from the $write$ stream by replacing each w event with the constant 5. In the second equation, the $delay$ function acts as a timer, which is set to the timeout value with the first argument and reset with any w event on the second argument. After 5 consecutive timestamps without a w event, an error is raised in the $output$ stream.

1.5 RUNTIME ENFORCEMENT

Runtime enforcement is a security enforcement paradigm that prevents a monitored program from misbehaving by intervening as needed to enforce a user-specified security policy [75]. Unlike RV, the monitor is intended to provide a proper alternative for any misbehaving trace rather than merely signalling a violation. The growing popularity of smart contracts has prompted fresh interest in runtime enforcement [56]. Smart contracts cannot be changed after they have been deployed. Therefore, the only way to deal with unexpected behavior is to enforce it at runtime.

Similar to runtime monitoring, the execution of a program is abstracted as a sequence of events, called actions, while the desired security policy is abstracted as a set of valid sequences, called the property. Enforcement is commonly performed by way of an enforcement monitor (EM): a formal model, such as a transducer, that receives as input the original sequence of the program and outputs an alternate execution sequence that provably respects the property. The EM is generally tasked with ensuring conformity with two basic principles: *soundness* and *transparency* [95, 163], where transparency states that if the original security policy was already valid, then the replacement sequence must be equivalent concerning some equivalence relation. In other words, the monitoring process cannot alter the semantics of valid traces. Further research also suggested that the monitor should be limited in the changes that it performs on invalid executions. Otherwise, almost any property can be enforced, but not necessarily in a manner that is useful or desirable [37, 113]. This observation bears a pivotal consequence on the conduct of the monitor: in cases where multiple possible paths to enforcing the property are possible, the monitor should be required to select the *optimal* one, with respect to some gradation of executions.

In this context, the monitor is actually a mathematical structure tasked with performing the entirety of the enforcement process: reading the input, transforming it through a process of substitutions, insertions, deletions, and/or truncations, ensuring compliance with the resulting output trace concerning both soundness and transparency, and ensuring that the output is optimal. The monitor must thus encapsulate the desired security policy, the desired gradation of the solution, and limitations (i.e. memory or computational limitations) imposed on the monitor's ability to transform input sequences. This monolithic design incurs several disadvantages, particularly causing difficulties in generating a correct monitor for a given security policy. In addition, elaborate proofs are often required to ensure that the output of the monitor is indeed sound and transparent (see e.g. [113]). A detailed discussion of existing works on enforcement will come in Chapter 2.

CHAPTER II
STATE OF THE ART

In this chapter, we present an overview of the current state of the art in runtime enforcement and runtime verification with a focus on access restrictions. We introduce the problem statement of this thesis, exploring the factors contributing to uncertainty in events, which serve as the motivation for our approach. In Section 2.3, we present an in-depth portrayal of the diverse approaches encountered in the existing literature, providing a comprehensive overview. This is followed by a comparative analysis in Section 2.4, where we examine and contrast these approaches to identify their similarities and differences..

2.1 PROBLEM STATEMENT

Despite the variety of event types and event sources that can be used for analysis, a widely held assumption in RV is that the monitor has complete and error-free access to the set of events against which to evaluate a given property [32, 97, 103]. However, it has been acknowledged that this assumption is not completely warranted, as there exist multiple situations where the monitor may operate with some level of uncertainty about the content of the underlying trace. In fact, a recent Dagstuhl seminar report has emphasized the importance of dealing with incomplete, imprecise, and faulty sources of events [16], as did a recent survey of challenges related to Runtime Verification [161].

An uncertain or missing event hampers the monitoring process, and the monitor may not be able to produce a conclusive verdict. On the other hand, ignoring this event may affect the monitoring result as it could be relevant to the property being monitored. The monitoring system will not be able to identify when the property is violated, resulting in undetected errors or security breaches going unnoticed. An alternative approach is to replace this event with all possible replacement events. However, this will result in multiple output traces and consequently multiple output verdicts rather than one possible verdict.

Obtaining several output traces raises the question of which one of them is the best. As we have seen, this problem of runtime verification is related to another problem of runtime enforcement [72, 150, 163]. Runtime enforcement complements runtime verification by additionally seeking to react to any observed violation in such a way as to correct and recover from it by modifying the execution or skipping execution steps.

One challenging problem is the quantitative enforcement [136], which amounts to find the "best" enforcement mechanism for a security policy and a target, in accordance to some quantitative criteria such as cost or precision of enforcement. Quantitative enforcement could be seen as an optimization problem to find the best among all good enforcement mechanisms. There are several ideas on how "quantities" can be used in the ambit of enforcement of security policies: it could be the cost of enforcement (i.e. not all security mechanisms/decisions come for free), the uncertainty (i.e. systems evolving in a real setting are prone to uncertain parameters that must be taken into account), and the optimization (i.e. it is not sufficient to define a "good" enforcement mechanisms, we might wish to find the "best" one) that outputs a correct trace with minimal efforts.

2.2 CAUSES OF UNCERTAINTY

As we have seen in the previous chapter, existing approaches to runtime verification make use of a large number of models for the representation of events, as well as the expression of properties. There are almost as many tools and models as there are combinations of traces and specification languages, and some works have even attempted to define conversions to go from one to another [142]. However, the vast majority of these approaches are underpinned by a fundamental assumption: the trace on which the monitoring is carried out is complete, and all the events it contains are exact and devoid of any error or uncertainty. Regardless of the condition to be evaluated and the notation used to represent it, the verdict produced by a monitor is reliable only if this crucial condition is respected.

Yet, one can easily imagine situations where the contents of a trace may not entirely be trusted: events may go missing, numerical measurements may carry an intrinsic uncertainty, etc. We shall group under the term “data restriction” any situation where an input trace is considered unreliable, regardless of the reason. As we shall see in Section 2.3, some works in the field of RV address the issue in different ways. However, before even describing how the problem can be tackled, it is appropriate to discuss the various ways in which an input trace can become incomplete or uncertain. In this section, we present a synthesis of the various causes for such partial information that have been invoked in the literature.

2.2.1 MECHANISMS OF DATA RESTRICTION

A first element that needs to be studied is the actual location in the monitoring process where data restriction takes place, and in what way this restriction affects the evaluation of a property on a trace. Figure 2.1 represents a general view of the situations where data restriction may happen. In this figure, D is the original or “perfect” version of a data object (i.e. the input trace), while D' is a degraded, modified, or otherwise “unreliable” version of D.

A monitor M can be viewed as a process that performs a read operation on the contents of the data object, which can be likened to a form of “query” Q. The result of this query R corresponds to the trace (or part of the trace) whose content is needed by the monitor. For example, one could view the access to each individual event of a trace as a form of query-response loop that the monitor needs to perform in order to evaluate a given property. The figure represents four situations that can occur with respect to data restrictions. Situation 0, on the left-hand side of the figure, corresponds to the case where no data restriction occurs. Monitor M_0 performs a read operation Q_0 on the contents of the data object D and obtains the exact value in response R_0.

In situation 1, on the right-hand side, the monitor does not access the original data object D, but rather its restricted version D'. The monitor can still freely query the restricted data object D' by sending the query Q_1 and receiving a response R_1. This situation is not only representative of cases of (unintentional) data corruption, but also of deliberate restrictions meant to prevent access to the original trace contents. For example, values in a data object may be subject to anonymization, or parts of the object may simply be deleted to avoid unauthorized access.

In situation 2, at the bottom of the figure, the monitor M_2 queries the data object, but the original query Q_2 is transformed into a less precise query Q'_2 – or blocked altogether. The monitor will receive the "correct" response R_2, but for the modified query Q'_2 which probably queries different or less precise information than Q_2. Access control policies can be a reason behind blocking the query in this situation. Finally, in situation 3, the roles of the query and response are reversed. The monitor can query whatever it wants, but the response may get transformed before reaching it. This situation is similar to situation 1; however, the modifications in this situation are applied to the output of the query and not to the data object itself.

Situations 1, 2, and 3 can have the same observable effect on the monitor: receiving imprecise data or even nothing at all. However, the difference lies in the mechanism by which uncertainty or imprecision is introduced.

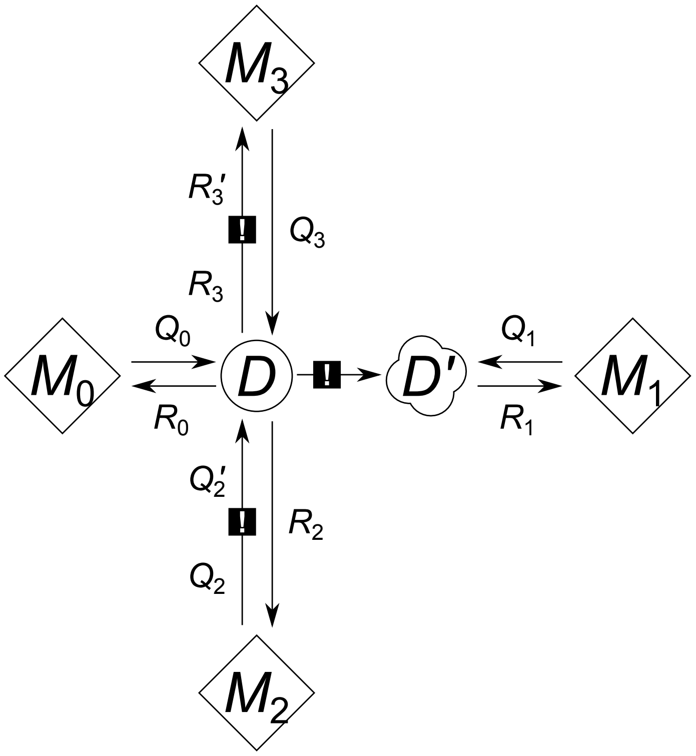


Figure 2.1 : An overview of access restriction situations.

Regardless of the mechanism by which data restriction occurs, causes of incomplete data in traces can be broadly divided into intentional and unintentional causes. Intentional causes are the restriction mechanisms enforced by the user; hence, they are expected, such as data restricted due to an access control policy. On the other hand, unintentional causes are unexpected phenomena that cause a loss of data, such as sudden data corruption. In a runtime monitoring context, both intentional and unintentional causes will affect the monitoring process due to their impact on the quantity and quality of the data available to monitor. In other words, some level of uncertainty will be introduced into the data fed to the monitor, which differs based on the method of restriction imposed.

2.2.2 INTENTIONAL CAUSES

We call "intentional causes" any deliberate operation that results in a degradation of the original input trace, which can impact the verdict returned by a monitor. For example, most information systems are equipped with mechanisms to prevent the disclosure of their confidential data. This can be achieved through access control mechanisms that determine who can access what, or by employing various data protection techniques such as data encryption and data anonymization. In both cases, access to the data is restricted, and some query responses may contain incorrect or incomplete data, or no response at all. We will now detail some of the possible intentional causes listed in the literature.

Intentional causes can be represented by any of the situations 1, 2, and 3 shown in Figure 2.1. In situation 1, intentional modifications can be applied to the data source D to obtain D'. The user then queries D' instead of D and receives an imprecise response containing data that differs from the original data in D. Similarly, intentional modifications can be applied to Q_2 (resp. R_3) in situation 2 (resp. 3) to obtain Q'_2 (resp. R'_3); the user will receive the result of Q'_2 (resp. R'_3) instead of Q_2 (resp. R_3).

Access-Control Policy An access control policy is a rule that defines who is authorized to access which data and under what circumstances he can do so. Several access control models are commonly used in computer systems [1, 8, 10, 52, 61, 62, 68, 102, 128, 133, 162, 176, 178, 190]. Each model has a different way of enforcing the rules, but they all aim to restrict access to certain data. Data encryption can also be seen as a mechanism for enforcing access control as it converts data from a readable format into an unreadable encoded format [43].

As a simple example, consider a log containing medical records where the "diagnosis result" field of each patient can only be accessed by a doctor. A runtime monitor verifying the satisfaction of a property such as "the number of patients diagnosed with cancer is equal to 60" would need to access the restricted data values to be able to compute the number of patients with cancer and produce a certain verdict. Without access to such data, the monitor will produce an inconclusive verdict.

Note that that the occurrence of access control policy restrictions is primarily associated with situation3. If Q_3 requests to access objects O_1 and O_2 while an access control policy states that the requester is permitted to only access O_1, the requester will receive an incomplete or reduced response R'_3 where the object O_2 is missing instead of receiving R_3.

Data Anonymization Data anonymization is a technique used to protect sensitive data by hiding personally identifiable information while maintaining the integrity of the data [58]. The process of data anonymization introduces uncertainty into data that was initially certain.

There are several data anonymization techniques [135]. *Generalization* consists of reducing the precision of attribute values by changing their scale. For example, a discrete numerical data value (such as age) can be replaced by an interval of values where one of these values is the correct original value (such as $[30-40]$). Similarly, a categorical data value (such as city name) in the original dataset can be replaced by a set of possible data values (such as {Montréal, Laval, Longueuil}). Each value in the dataset or data interval is considered as one possible "world". When a monitor accesses certain data values from the anonymized dataset, it will receive a set of possible events instead of one precise event.

Suppression is another anonymization technique, which completely deletes a data attribute or a part of the dataset. On the other hand, *replacement* is a method that involves substituting characters of an attribute or value in the data with a predefined symbol (such as X or *). It's important to note that in the case of complete suppression of a data value, the monitor will receive a missing event. In the case of partially masking characters, the monitor will receive partially incomplete events (uncertain events where a part of it is missing). Another type of anonymization involves slightly modifying data attributes by adding some random noise to make them less accurate. For example, adding or subtracting days or months from a date. In this case, the monitor will receive incorrect or corrupted events.

Data Perturbation While data anonymization techniques aim to remove or mask identifying information from the dataset to prevent linking the data back to specific individuals or entities, data perturbation techniques seek to balance the need for privacy protection with the need to maintain the usefulness and accuracy of the data for analysis and decision-making [50, 186]. There are various techniques for data perturbation, including randomization and noise addition, and data swapping.

One method of randomization is called projection perturbation [51], which is a geometric data perturbation technique applied to a dataset where values are represented as data points in a multidimensional space. A set of data points is projected from the original multidimensional space to another randomly chosen space. Another perturbation method is noise addition [143, 146, 159] where a certain amount of random noise can be added while still effectively reconstructing specific information, such as column distribution, from the perturbed data. For example, in the case of an execution log, suppose a sequence of events consists of successive numerical values $x_1, x_2, \ldots, x_n$. One way of applying data perturbation would be to modify the original data by adding random noise values $\overline{r} = r_1, \ldots, r_n$ to the original data, resulting in a modified trace $x_1 + r_1, x_2 + r_2, \ldots, x_n + r_n$. This modified trace would be published along with the distribution of $\overline{r}$. Such perturbation makes it impossible to recover the original content of each event but still preserves the validity of coarse-grained properties that apply to the set of values. For example, a property expressing a condition on the average of the events is likely to produce the same verdict on the original and the modified trace.

Data perturbation by swapping involves exchanging or swapping data values. In runtime monitoring, this could be by done swapping data values of an event attributes or swapping entire events in a trace. For example, in a traffic light dataset, if the green light event is swapped with the yellow light event and a monitor is checking the property *a red light should always be followed by a green light,* the resulting verdicts will be imprecise due to the swapped events.

Load Shedding and Throttling In many cases, feeding an event to a monitor requires additional work for the executing system. For example, it is well-known that in the runtime verification of Java programs, weaving AspectJ pointcuts and making repeated calls to multiple monitor instances introduce a non-negligible overhead, especially in terms of execution time. In other cases, events can be stored in a database (relational or otherwise), and a monitor may periodically query this database for any new incoming events. This is the case, for example, in environments with embedded devices like smart homes [80]. Executing such a query inevitably adds an extra load on the system. A similar situation may arise for monitors whose source of events is web services or other forms of web-based implementations, where events typically take the form of HTTP requests or responses [89]. In all these situations, the simple act of feeding an event to the monitor comes with a cost. Monitoring a property can therefore strain a system beyond its available resources. In such cases, some systems may intentionally impose a usage limit. For example, the Amazon Marketplace Web Service (AMWS) imposes a quota of 720 requests per hour for operations like `List-Matching-Products` [12]. Similarly, BizTalk Server limits the number of messages that can be sent or received [139].

One possible solution is to implement *load shedding*, which involves deliberately discarding events in order to reduce resource consumption. For instance, Joshi *et al.* [105] describe a scenario where a media player software is instrumented with a library that operates within a fixed time budget. In a given time interval T, the instrumentation can generate a maximum of B events; any event exceeding this threshold within the interval is replaced by a special “non-event” called χ.

Load shedding is predominantly applied in data stream management systems, where processing delay is a critical quality metric. In cases of overload, which are common in data stream systems, the ability to maintain a desired level of delay is severely limited. While load shedding reduces overhead and allows processing time to keep up with the rate of incoming inputs during overload situations [173], the resulting datasets after load shedding can have varying levels of accuracy due to missing data values [144, 174].

Data Sampling Sampling is a technique used to systematically select a subset of data values from a pre-defined population to serve as a data source for data analysis tools and runtime verification monitors [138]. Sampling techniques can be broadly divided into two categories: probability and non-probability sampling. In probability sampling, one can specify the probability of an element (such as events with the attribute x equal to a certain value n) being included in the sample. Among the probability sampling techniques, we have “simple random sampling,” where each element has an equal chance of being selected, and “stratified random sampling,” where each element has a known probability of being selected.

In non-probability sampling, the probability of including an element in the sample cannot be estimated. Hence, it is less precise than probability sampling but also less expensive. Among non-probability sampling techniques, we have “quota sampling,” where quotas are set for the number of elements to be included in the sample based on certain characteristics. These quotas are determined based on prior knowledge of the population. Another technique is “convenience sampling,” which involves selecting sample units based on their accessibility to the selector. Factors such as geographic proximity, availability during the study period, or willingness to participate in the analysis can influence the selection.

It is important to note that the monitoring result depends on the precision of the sampling technique. In other words, by selecting a representative subset of the trace for analysis, the runtime verification process can be made more efficient while still providing a high degree of confidence in the correctness of the system's behavior. Data sampling has been used to mitigate computational overhead in runtime monitoring [42]. Arnold *et al.* [14] presented a runtime environment that can efficiently check violations of user-specified correctness properties with controlled overhead. They introduced property-guided sampling, specifically object-centric sampling, to collect sampled profiles while preserving analysis correctness. Property-guided sampling ensures that the sampled profile maintains sufficient properties to make the dynamic analysis meaningful. Object-centric sampling allows the analysis to sample at the object instance level. An object can be marked as tracked, and the analysis can receive all profile events for this object while receiving no events for untracked objects.

In other monitoring approaches, sampling is achieved by temporarily disabling the monitoring process. This is the case with Huang *et al.* [104], whose proposed technique temporarily disables monitoring of selected events for the shortest possible duration while ensuring that the user-specified target overhead is not exceeded. Fei *et al.*'s [76] method selectively enables monitoring for specific function executions. By default, their method tracks a function execution only if it is called in a previously unseen context. Theoretically, a function's context encompasses all memory locations it accesses. Storing and comparing all such contexts would be prohibitively costly. They use less demanding definitions of "context" and "context matching," which may result in missing some interesting behaviors.

2.2.3 NON-INTENTIONAL CAUSES

Apart from the intentional causes enumerated above, there are also unforeseen situations that result in data restriction. All non-intentional causes belong to situation 1 of Figure 2.1, where the data source D is changed to D' after applying certain modification technique(s). The user will query D' instead of D and receive an imprecise response containing data different from the original data existing in D.

Data Corruption A first obvious non-intentional cause is data corruption. Events in a trace can be stored on a medium that degrades over time and may render access to some of their values impossible. Error detection codes, such as CRC-32, can also reveal that stored data is invalid without necessarily providing the means to recover the original data. In such a situation, all one can know is that an event occurred or that some value was recorded, but the actual contents cannot be trusted.

Another common type of data corruption occurs during data transmission when a data event or an interval of events is dropped from the stream. This can happen, for example, due to a momentary communication link failure or as a result of environmental factors interfering with data transmission, particularly when using wireless transmission methods. Assuming that each transmitted data value is assigned a unique and incrementing ID, the presence of non-successive IDs can be used by a user connected to the source where these events are stored to detect the occurrence of such a drop. This makes it possible to determine how many events occurred, but not their values.

Incorrect System Instrumentation As mentioned in Section 1.2, instrumentation is a computational process that extracts and records events from a software system during execution to make them available for analysis by a decision procedure, such as an RV monitor [25]. The recorded events are sent to the monitor as an ordered stream (a trace of events). The event order in the execution trace is usually guaranteed by instrumentation to correspond to the order in which the appropriate computing step occurred. However, in some cases, such as distributed environments, only a partial ordering of events can be properly relayed to the monitor.

There are other situations where logging statements are manually inserted by the developers [187, 188]. In such a context, many relevant logging statements can be missing from a system [189]. Each logging statement is typically assigned a log level. There are typically six types of log levels ordered based on their verbosity: `TRACE` > `DEBUG` > `INFO` > `WARN` > `ERROR` > `FATAL`. The usage of these levels by developers can be highly unreliable [126], where the same statement in two distinct code locations can be assigned two different levels (e.g., `INFO` vs. `DEBUG`). For example, if a user sets the verbosity level to be printed at the `WARN` level, only the logging statements with the level `WARN`, `ERROR`, or `FATAL` would be printed out (and thus reach a monitor). If a relevant event for the evaluation of a property is assigned the incorrect level, it runs the risk of being filtered out based on the verbosity level and not reach the monitor.

Such manually-generated logging statements can also be imprecise in themselves. For example, suppose that a message such as “Error reading resource” can be used to indicate either a disk or a network failure. A monitor for a property such as “every disk failure must stop the program” may report incorrect violations because two types of failure get the same message and translate into the same event.

Imprecise Measurements In some situations, events may represent values measured by sensor devices that may suffer from low data quality due to long-term use and other environmental factors [130], resulting in bias, drifting, full failure, or precision loss, and other faults in the data recording process. Supplying a decision procedure such that a runtime monitor with inaccurate data from sensors will affect the verdict produced by the monitor. For example, consider a sensor recording temperature producing a value T having an error range, e.g., $T = 20° \pm 0.5$. If the verdict produced when monitoring a property depends on whether $T \leq 20$ or $T > 20$, the monitor will not be able to produce a definite verdict for a range of values of T.

Another example of such a situation is illustrated by the monitoring of the position of a drone [177]. The altitude a of the drone can be modeled as a probability distribution. In such a model, a Boolean statement such as $a > 3$ cannot be expressed directly, as the precise value of a is unknown. One can only speak of the probability $Pr(a > 3)$; in such properties, Boolean statements are recovered by giving bounds, such as $Pr(a > 3) \geq 0.99$.

Impedance Mismatch Impedance mismatch is a cause of data uncertainty that occurs while checking a property over a trace of events during runtime verification. In order to monitor an event, the property is checked over the event parameters and emits a verdict if the parameters of the event are compatible with the property. Usually, we can solve this issue by rewriting the property so that it can align with the instrumentation and the event parameters. Impedance mismatch occurs if two conditions are satisfied: first, there is no knowledge about event parameters. Second, the parameters used to express the property do not align with the event parameters, and there is no possibility to rewrite the property so that it matches the event parameters.

For example, a property may specify conditions on individual values of x and y, while the source of events only gives their sum s. Impedance mismatch can occur, for instance, when one wishes to monitor a new property over a log that has been recorded for another purpose. One solution that does not require rewriting the property is to turn values of s into imprecise versions of x and y.

2.2.4 EFFECTS OF DATA RESTRICTIONS

In the preceding sections, we described the mechanisms of data restrictions and all the causes of data restrictions that can happen intentionally and non-intentionally. We hinted by means of a few examples to the impact that these restrictions can have on the verdict produced by a monitor. In this section, we examine this notion in more detail and discuss the possible effects of data restrictions on the monitoring process.

Consider a simple situation where possible atomic events are $\Sigma = \{a, b, c\}$, a trace $\overline{\sigma} = abcababbca$, and the simple property that stipulates that "every a must immediately be followed by b". If the monitor is fed event a, and the subsequent event b is dropped from the trace, it will incorrectly conclude that the property is violated upon receiving the next event c. The same will happen if, instead of being dropped, b is corrupted and turned into event c. In those situations, the monitor reaches a definitive verdict, but this verdict is incorrect in light of the content of the original trace.

A different set of issues can arise if the presence or content of events is uncertain. For example, suppose that the actual identity of the second event of the trace is not known. In such a situation, the monitor cannot reach a definitive verdict: the property could be satisfied (if the unknown event is b) or violated (if it is anything else). A similar outcome occurs in the situation where b may or may not have occurred.

We distinguish between eight types of data restrictions. This categorization will be used in later sections to classify the works on runtime verification under uncertainty according to the type of restriction they consider.

Case 1: We know exactly one event is missing and where In this first case, the monitor is given a trace where the number and location of missing events is known. For example, a monitor might receive the input trace $\overline{\sigma} = abc\chi babbca$, where χ is marker indicating that at this precise location, an event is known to have occurred but was lost. We have seen this happens in some cases of load shedding where actual events are dropped and replaced by an empty “non-event”. This can also occur in situations where each event is given a sequential number, and where a gap in the order of these numbers is detected.

In such a situation, the monitor may ignore the missing events and proceed with the next event, or generate a non-conclusive verdict, or consider the set of all possible events that may occur in this gap. Note that this case can be extended to the situation where n successive events are known to be lost (which would be detected by the presence of multiple successive χ markers), or multiple individual events are missing throughout the input trace.

Case 2: We know an event is invalid, but we can’t recover its contents This case is handled in the same way as the previous one. A corrupted event can be considered as a missing event whose occurrence is known. As we discussed earlier, corruption can be made known by means of checksums and other integrity checks, which can typically uncover the presence of corruption but not always recover from it.

Case 3: We know events are missing, but we do not know how many This time the χ marker may only be interpreted as the presence of an *interval* of missing events, but the number of events in this interval is unknown. Thus a runtime monitor may receive the trace $\overline{\sigma} = abc?abbca$, where this time “?” indicates the location of an interval of missing events. This could occur, for example, when the communication link feeding events to the monitor is interrupted and then resumed, but without the presence of sequential numbers that could indicate how many events have been lost in the meantime.

This case is much harder to handle than the previous two, due to the higher degree of uncertainty on the contents of the trace. Yet, in some cases, a monitor can still recover from such situations and produce a sound verdict. For example, if the monitor evaluates the property "every c is eventually followed by an a", it could conclude that the received prefix satisfies the property regardless of the length and content of the missing gap.

Case 4: Events are missing and we don't know about it In this case, no marker is even present to signal possibly missing events. Thus, a monitor would receive for example the trace $\overline{\sigma} = abcabbca$; the monitor is not notified of whether, if any, and where, are missing events in this input trace. As with case 3, a monitor could still produce a valid verdict for some input traces and some properties, however it does not even have a mean of knowing when its verdict could be incorrect. We list this situation for the sake of completion, but it goes without saying that none of the surveyed works address this situation.

Case 5: Events are corrupted and we don't know about it This is equivalent to case 4. The monitor will process the event as if ignoring the presence of corrupted events.

Case 6: We know an event may be one from a set, but we don't know which one This can be seen as a more precise type of uncertainty than cases 1 and 2. Instead of supposing that a missing event could be any one in Σ, this time the monitor is given slightly more precise information as a set of possible events is known. One solution could be replacing the event with a set of possible replacements or by the conjunction of the elements of this set. For example, the monitor could receive the trace $\overline{\sigma} = abc\{b, c\}babbca$, where the exact value of the fourth event is unknown, but it can only be b or c.

This happens, for example, if an event is partially corrupted, so that its contents is known in part (enough to eliminate a set of possibilities over what it could be). It is also a symbolic way of representing uncertainty over numerical values; thus a value of 20 ± 0.5 indicates that the "true" value can be any one in the interval $[19.5, 20.5]$, without knowing exactly which one it is.

Case 7: We know an event X may or may not have occurred In this situation, the monitor is fed events, but some of them have a marker indicating that their occurrence is uncertain. For example, a monitor could receive a trace $\overline{\sigma} = ab\dot{c}abbca$, where the dot over the first c indicates that this event may or may not have occurred. Conceptually, this case can be handled in a way similar to Case 6, if one allows the empty event ϵ to be one of the possibilities.

Case 8: We know events X and Y occurred, but we don't know which came first This situation happens in cases where the interleaving of multiple events is not precisely known, such as in the Life Data Recorder discussed earlier. In this case, the monitor could receive a trace such as $\overline{\sigma} = ab(c||a)bbca$, where $c||a$ indicates that both c and a have occurred, but their exact ordering is missing. The monitor in this case could consider the two possibilities $\{ca, ac\}$ and produce a set of two possible verdicts.

We shall mention that, depending on the specification property, the monitor may be able to produce a conclusive verdict regardless of what and how many the missing events are. For example, if the property states that each b should be finally followed by a, once receiving the event a after the gap, the monitor is able to produce a conclusive and sound verdict. Moreover, in some situations and for some specification properties, extending an existing specification language with useful operators allows writing a specification property in a way that avoids the need for the missing or uncertain event in producing a correct verdict [27].

2.3 EXISTING RV APPROACHES TO DATA RESTRICTIONS

As explained in Section 2.2, the presence of data restrictions can be caused by a variety of factors, either intentional or unintentional. Moreover, data restrictions obviously have an impact on the verdict produced by a monitor in some situations. In this section, we survey and categorize the various approaches that have been taken in runtime verification literature to address this issue. Two types of data restrictions must be distinguished at the onset: *unknown* restrictions correspond to the first example, where the monitor has no means of assessing where and how data restriction occurs; for example, when an event is dropped and no mechanism exists to inform the monitor of its absence, or when its contents are corrupted and it is impossible to discover this. By definition, it is impossible to *always* recover from such a type of data restriction: the monitor will necessarily produce an incorrect verdict in some situations; In other words, will not be able to produce a single conclusive verdict.

Consequently, the works surveyed in this section rather address *known* data restrictions. These correspond to alterations of the input trace that the monitor is made aware of. Examples of this type of uncertainty include: a numerical measurement accompanied by an interval of uncertainty (such as temperature measurement sensor affected by an uncertainty of ±2 degree); a placeholder indicating that an event occurred without knowledge of its actual contents; or a mechanism that can identify that a data object is corrupted without the capability of recovering its contents. In those cases, a monitor can warn its user that the presence of data restrictions may have an impact on the accuracy or the validity of its verdict.

The nature of this warning varies from one study to another: some works propose a verdict associated to a probability; other works output multiple possible verdicts. As we shall see, some approaches use statistical methods to create a runtime verification model capable of computing a final verdict, while others build the model using formal languages and automata theory, and many approaches work on the abstraction of incomplete event traces to achieve an abstract verdict.

In this section, we describe approaches from literature that tackle the problem of runtime verification with incomplete or imprecise data and we classify them into abstraction based approaches, statistical-based approaches and language-based approaches. Each of these approach is described in the same way, by summarizing the following elements:

- The type of uncertainty targeted, by linking them to the various cases enumerated in Section 2.2.4
- The type of events (atomic, numerical, tuples, etc.) and the way uncertainty about them is represented
- The method used to represent the specification and the type of verdict produced by the monitor (e.g. probability, interval, set of possible values, etc.)

The formalism used by each approach to represent the events and the specification properties are listed in Table 2.4.

Furthermore, we divide the existing works into three broad families of techniques: abstraction-based approaches (§2.3.1), language-based approaches (§2.3.2), and statistical-based approaches (§2.3.3).

2.3.1 ABSTRACTION-BASED SOLUTIONS

Some RV approaches use abstraction methods to solve the problem of monitoring a property over an incomplete trace. Attempting to fill a gap in a trace with all possible replacements for the missing or uncertain event will produce a large number of concrete traces.

Abstracting the set of concrete traces into one abstract trace will simplify the approach. In this section, we discuss the RV approaches based on abstraction.

2.3.1.1 LEUCKER *ET AL.* [124]: RUNTIME VERIFICATION FOR TIMED EVENT STREAMS WITH PARTIAL INFORMATION

Type of uncertainty targeted For their part, Leucker *et al.* proposed a solution for runtime verification over streams of data containing missing and imprecise values. A data stream is a sequence of timestamps and data values representing the stream's events. To model imprecise values, streams are lifted from concrete domains of data to abstract domains. For example, a concrete numerical value in a concrete stream can be represented as an interval of real numbers in the abstract stream. Briefly, an abstract event stream is represented as multiple concrete event streams carrying information about the events and the gaps (this represents cases 6 and 7 of Section 2.2.4).

Type of events and their representation With respect to event representation, a concrete event at a timestamp t can be a known event d of any type (such as Boolean) belonging to a data domain $\mathbb{D}$, $\perp$ if there is no event at t, or "?" for timestamps after the progress of the stream. A data abstraction of a data domain $\mathbb{D}$ is an abstract domain $\mathbb{D}^\#$ where a particular point t can either be a known event from $\mathbb{D}$ with a known timestamp, $\perp$ if there is no event at t (but there are events at $t' > t$), $\top$ if there is an event at t but it is unknown (imprecise), and $\smile$ to represent a gap (a segment of an abstract event stream that represents all combinations of events that could possibly occur in that segment, both in terms of timestamps and values).

Method used to represent specification property and the verdict type Leucker *et al.* extended the TeSSLa specification language described in Section 1.4 into *Abstract TeSSLa* by defining an abstract counterpart operator for each concrete operator of TeSSLa. This allows deriving an abstract specification property from a concrete specification property by replacing every concrete TeSSLa operator with its abstract counterpart.

The abstract specification is proved to be a sound abstraction of the concrete specification, i.e., every concrete verdict generated by the original specification on a set S of possible input traces is represented by the abstract verdict applied to an abstraction of S. For example, in the domain $\mathbb{B}$, a concrete event can be $true$, $false$, or $\perp$. Applying a concrete TeSSLa specification, we get a conclusive concrete verdict for the $true$ and $false$ events, and a non-conclusive verdict when encountering $\perp$ (missing event) or an imprecise event or a gap of any length. However, for an abstract trace in the domain $\mathbb{B}^{\#}$, an abstract verdict (set of concrete verdicts) is produced when applying the abstract specification over the abstract events. For a known event, the resulting abstract verdict contains one concrete conclusive verdict. For a missing event, which is still represented as $\perp$, the abstract verdict is the same as the verdict produced on a concrete event $\perp$. For an imprecise event replaced by $\top$ which represents any possible event from $\mathbb{B}$, the abstract verdict is a set containing all the possible conclusive verdicts. For a gap replaced by $\smile$, the abstract verdict is a set of all possible conclusive and non-conclusive verdicts.

2.3.1.2 WANG *ET AL.* [184]: RUNTIME VERIFICATION OF TRACES UNDER RECORDING UNCERTAINTY

Type of uncertainty targeted Wang *et al.* [184] present an approach for runtime verification to handle the uncertainty that arises due to imprecise order of events in a trace. A Life Data

Recorder device (LDR) is used to collect updates to data variables such as x and y and stores their values as a snapshot vector (§1.3) or a frame in the memory. Some variables are process variables that are updated once in a frame, while other variables can be updated several times in the same frame. Uncertainty arises when the update of one variable interleaves with the other variables in the same frame and the knowledge about the exact ordering of their updates in the frame is lost. As a simple example, suppose that a process variable x is updated one time (from value 2 to value 3) in the frame f, and the variable y is updated two times (from value 4 to 3 and from 3 to 5) in the same frame f. One possible ordering of (x, y) updates could be $(2, 4) \xrightarrow{\text{x}} (3, 4) \xrightarrow{\text{y}} (3, 3) \xrightarrow{\text{y}} (3, 5)$. When x and y interleave, we cannot determine the exact ordering of (x, y) updates (this represents case 8 of Section 2.2.4).

Type of events and their representation Wang *et al.* consider each frame recorded by LDR as an abstract state, and each mapping from the variables x and y to their values a concrete state. A possible concrete state is (2,4) which maps x to value 2 and y to value 4. Several traces of concrete states can be extracted from one abstract state such as:

$$(2, 4) \xrightarrow{\text{x}} (3, 4) \xrightarrow{\text{y}} (3, 3) \xrightarrow{\text{y}} (3, 5)$$
$$(2, 4) \xrightarrow{\text{y}} (2, 3) \xrightarrow{\text{x}} (3, 3) \xrightarrow{\text{y}} (3, 5)$$
$$(2, 4) \xrightarrow{\text{y}} (2, 3) \xrightarrow{\text{y}} (2, 5) \xrightarrow{\text{x}} (3, 5)$$

A sequence of abstract states form an abstract trace Tr. The set of concrete traces consistent with the abstract state $Tr(i)$ is represented as $Path(Tr(i))$.

Method used to represent specification property and the verdict type Past LTL (§1.4) [97, 134, 152] is used to represent the monitoring property using atomic formulas such as $\odot\varphi$

(meaning that φ was true at the immediately previous state), $\diamond\varphi$ (meaning that there was some time in the past when φ was true), $\boxdot\varphi$ (meaning that φ was always true in the past), and $\phi S\psi$ (meaning that either ϕ was always true in the past, or ψ held somewhere in the past and since then ϕ has always been true).

Wang *et al.* aim to monitor a property over the abstract trace provided by the LDR. They keep the syntax for past-LTL and introduce a new three-valued semantics based on standard semantics for concrete traces. A formula φ evaluates to true ($\top$) on an abstract trace Tr only if it evaluates to $\top$ on all concrete traces consistent with Tr; it evaluates to false ($\perp$) on Tr only if it is $\perp$ on every concrete trace consistent with Tr; otherwise a non-conclusive "?" is resulted. To monitor the property $\varphi = \phi S\psi$ over a concrete trace $p_0, ..., p_m$ a checking algorithm iterates through all concrete states from p_0 through p_m. In each concrete state p_j, the checker keeps the resulting verdicts of all subformulas (ϕ and ψ) on the trace $p_0, ..., p_{i-1}$ (called the checker state). The checker updates its state based on the values in p_i.

To monitor the formula $\varphi = \phi\,\mathbf{S}\,\psi$ over an abstract trace Tr, the semantics are built in a recursive fashion assuming the resulting verdicts of checking the subformulas ϕ and ψ over the partial trace $Tr(i)$ is finished and available in a mapping $SV_i : SubFormulas(\varphi) \rightarrow \{\top, \perp, ?\}$. The function *checkOne*$(SV_i; p; \varphi)$ is then used, where p is one concrete trace from $Path(Tr(i + 1))$, the function returns whether φ is satisfied on all, none, or some (neither all nor none) concrete traces formed by concatenating any concrete trace in $Path(Tr(i))$ with p.

Kallwies *et al.* [108] studied the problem of recurrent monitoring with partial knowledge about input events. Recurrent monitoring checks a property from a specific position t in the trace (not necessarily a prefix of the trace). Each event is represented as a tuple of atomic symbol and position in trace. If a violation of the property occurred, it is associated with this particular position t rather than the entire trace. Kallwies *et al.* extended recurrent monitoring to k-offset recurrent monitoring where the verdict that the monitor must compute is shifted by a constant offset k. They extend past-LTL with bounded future and propose anticipatory recurrent monitoring. The anticipatory monitor computes functions that predict the future verdicts of the original monitor which are possible after the current observation. It can be also used to handle uncertain events. An uncertain input event is modeled as a set of possible inputs that actually happen. Kallwies *et al.* also used assumptions to improve the anticipation. Another approach for Kallwies that deals with uncertainty using assumptions is in [107].

2.3.2 LANGUAGE-BASED SOLUTIONS

Aside from statistical and abstraction-based methods, some approaches proposed a formal language equipped with useful operators to write a specification property that can produce conclusive verdicts when monitoring a trace with incomplete events.

2.3.2.1 JOSHI *ET AL.* [105]: RUNTIME VERIFICATION OF LTL ON LOSSY TRACES

Type of uncertainty targeted They presented an approach to the problem of RV in the presence of transient loss, which is a non-permanent loss of an event or a finite sequence of events is lost in a trace. After the data loss, the number of events that happened is known but their content is unknown (this represents cases 1 and 2 of Section 2.2.4). The goal of the authors is to show that there are some properties that can be monitored regardless of the presence of lossy events, under the condition that the monitor is able to observe subsequent valid events after the loss.

Type of events and their representation An event can be a single atomic proposition from an alphabet Σ or an atomic formula composed of atomic propositions connected using Boolean operators (such as conjunction $\vee$ and disjunction $\wedge$). A lossy event is represented by the symbol χ.

Method used to represent specification property and the verdict type The specification property is expressed using LTL (§1.4) and converted into an RV-LTL monitor, which is a finite-state automaton presented by Bauer *et al.* [32] as an extension of the LTL_3 semantics into $\mathbb{B}_4 = \{\top, \top_p, \bot_p, \bot\}$, where $\top_p$ and $\bot_p$ are emitted whenever an observed system behavior has not yet lead to a violation or acceptance of the monitored property. The value $\top_p$ (respectively $\bot_p$) means that the system will *presumably* satisfy (respectively violate) the property in the future. In order to determine whether the property is monitorable over a lossy trace, Joshi *et al.* build an algorithm that searches for a loss-tolerant alphabet and a loss-tolerant cluster in the RV-LTL monitor. A loss-tolerant alphabet represents the input elements where each element forces the monitor to transition into a unique state irrespective of its current state. The monitor is supposed to move to the unique state at the end of the loss if the processed element after the loss belongs to the loss-tolerant alphabet. A loss-tolerant cluster constitutes the set of states where each state transits the monitor to the same next state within the cluster when processing the same input from the loss-tolerant alphabet. Hence, If a loss occurs when the current state of the monitor belongs to a loss-tolerant cluster, the transitions of the cluster ensure that the next state would still be one of the same cluster.

A loss-tolerant monitor $\mathcal{M}$ is derived from an RV-LTL monitor by adding a new state. Whenever a lossy element χ appears, the monitor moves to this state and outputs the verdict "?". Hence, a loss-tolerant monitor produces an output in the truth-domain $\mathbb{B}_5 = \{\top, \top_p, ?, \bot_p, \bot\}$ which is $\mathbb{B}_4$ augmented with "?".

2.3.2.2 BASIN *ET AL.* [27]: MONITORING COMPLIANCE POLICIES OVER INCOMPLETE AND DISAGREEING LOGS

Type of uncertainty targeted They study the effect on RV of missing data due to logging failures and disagreement between logs about the occurrence of certain events when multiple logs are required to verify a property.

Type of events and their representation Basin *et al.* represent the uncertainty over event occurrences by means of what they call a *logging knowledge base*. A knowledge base is a sequence $\overline{\mathcal{D}} = \mathcal{D}_0, \mathcal{D}_1, \ldots$ of first-order structures defined over the set of ternary Boolean values $\{\top, \bot, ?\}$, where "?" represents the unknown truth value. Each first-order structure represents a discrete time point, and totally defines the (ternary) truth value of each event predicate. Informally, for some predicate r of input arity n, $r(a_1, \ldots, a_n) = \top$ in a given time point τ indicates that the event $r(a_1, \ldots, a_n)$ with parameters $a_1, \ldots, a_n$ happened at τ. Conversely, $r(a_1, \ldots, a_n) = \bot$ in a given time point τ indicates that the event $r(a_1, \ldots, a_n)$ did not happen at τ. Finally, $r(a_1, \ldots, a_n) =?$ represents a *knowledge gap* with regard to whether $r(a_1, \ldots, a_n)$ happened at τ (this represents case 7 of Section 2.2.4).

Method used to represent specification property and the verdict type Basin *et al.* propose what they call a compliance policy language L_3, which is a variant of First-Order Temporal Logic (FOTL) [26], to formalize and evaluate compliance policies in the presence of incomplete knowledge. A compliance policy is typically represented as a set of regulative normative statements (norms), that express what conditions need to be held by an agent to be authorized to do specific actions. Norms are applied at all times within a system, and deadlines are critical to manage temporal norms. Based on these notions, a compliance policy in L_3 is a closed formula of the form $\Box\forall\bar{x}.\varphi$.

For logging failure, Basin *et al.* assume that during the logging process, all events at are recorded correctly, and if a logging failure happens at a time point τ, the logging process stops and nothing is recorded until the process is restarted. Based on this assumption, policy violation could be avoided at τ (where the failure happens) if the policy to be checked depends on the past events that are already recorded before τ. The language L_3 is equipped with the temporal connective operator $\blacklozenge_{[b,b')}\varphi$ which returns $true$ if φ is $true$ at least at one past time point in the time interval $[max(0, \tau - b^{'} - 1), \tau - b]$, and $false$ if it is $false$ at all the time points in this interval. For example, the compliance policy *If a request is serviced at a web-server, then it must not have been denied by a firewall (in the past x time points)* is formalized as $\Box\forall r.(\text{service}(r) \longrightarrow \neg\blacklozenge_{[0,x)}\text{deny}(r))$, where r is the request and $\text{service}(r)$ and $\text{deny}(r)$ are predicates respectively representing the servicing and denying events of the request r. If the failure happens at τ and we want to verify the predicate $\text{service}(r)$ at τ, then all requests that had been denied at the previous x time points potentially violate the policy. However, if none of these time points has $\text{deny}(r)$ hold, the policy is therefore satisfied. So, not all logging failures must result in potential violations.

L_3 also specifies the obligations that should be respected by two parties exchanging documents; for example, the policy "all received documents must be paid for within 5 days". L_3 provides the operator $\Diamond_{[0,6)}pay(d)$ which has the same interpretation as $\blacklozenge_{[b,b')}\varphi$ but for future time points, and the operator $\otimes$ which is better than $\vee$ and $\wedge$ in the sense that none of the parties will be favored over the other when they disagree about the occurrence of an event. The policy is stated in L_3 as follows: $\Box\forall d.send(d) \otimes receive(d) \longrightarrow \Diamond_{[0,6)}pay(d)$. If nothing is sent at τ, the receiver's log does not contain a receive(d) and the sender's log does not contain a send(d), then the receiver in this case will not pay anything ($false \otimes false \longrightarrow false$ evaluates to $true$ meaning that the policy is satisfied). Contrarily, when a document is sent, we have that $true \otimes true \longrightarrow true$ evaluates to $true$ meaning that the policy is also satisfied. However, the sender may insert fictitious send(d) events to oblige the receiver to pay while the receiver's log disagrees (no receive(d) event in the receiver's log). In this case, the $\otimes$ operator can be used: $true \otimes false$ evaluates to $\perp$, and $\perp \longrightarrow false$ evaluates to $\perp$. In this case, specification no longer favors one party over the other.

2.3.2.3 BASIN *ET AL.* [28]: ON REAL-TIME MONITORING WITH IMPRECISE TIMESTAMPS

Type of uncertainty targeted Basin *et al.* raised the problem of imprecise timestamps of traces influencing the correct verification of the properties.

Type of events and their representation Two types of traces are considered: an observed trace and a real trace. The observed trace is a timed word $\overline{\sigma}$ containing imprecise timestamps and is represented as a sequence of tuples (τ_i, a_i) where $i \in \mathbb{N}, \tau_i \in \mathbb{T}$ ($\mathbb{T}$ is a discrete time domain) is the time stamp and $a_i \in 2^P$ is an atomic proposition from P. The real system trace, which contains precise timestamps, is represented as a timeline $\rho_{\overline{\sigma}}$.

To represent the imprecise timestamps, Basin *et al.* assume a timestamp imprecision $\delta \geq 0$, where an imprecise timestamp is assumed to belong to $[\tau_i - \delta, \tau_i + \delta]$ (this represents case 6 of Section 2.2.4 applied to timestamps instead of event values). A set of timelines $TL(\overline{\sigma})$ can be obtained from a timed word based on the function $\pi : \mathbb{T} \longrightarrow 2^P$, where $\pi(t) = a_i$ if $ts^{-1} = \{i\}$ (where $ts : \mathbb{N} \longrightarrow \mathbb{T}$ is an injective function and $ts(i) \in [\tau_i - \delta, \tau_i + \delta]$) or $\pi(t) = \emptyset$ otherwise). For example, if $\overline{\sigma} = (\{p\}, 1), (\{q\}, 1), (\{r\}, 2), (\{s\}, 5)...$ and $\delta = 1$, the time intervals are $[0, 2], [0, 2], [1, 3], [4, 6]$ and a possible timeline is π where $\pi(0.6) = \{q\}, \pi(1.2) = \{r\}, \pi(1.3) = \{p\}$ and $\pi(t) = \emptyset$ for $t \in [0, 4) \backslash \{0.6, 1.2, 1.3\}$.

Method used to represent specification property and the verdict type Basin *et al.* use MTL (§1.4) to rewrite φ into $tf(\varphi)$, where $tf(\varphi)$ accounts for timestamp imprecision by relaxing the implicit temporal constraints on atoms. For example, instead of having "p holds now", we have "p holds at a time point within the interval $[0, \delta]$ in the past starting from now or p will eventually hold at a time point within the interval $[0, \delta]$ from now". Formally, $p \in P : tf(p) := (\blacklozenge_{[0,\delta]} p) \vee (\lozenge_{[0,\delta]} p)$. On the other hand, they transform $\overline{\sigma}$ into a $monitored$ timeline $\rho_{\overline{\sigma}}$ by ignoring timestamp imprecision. Then they use an existing monitor (for precisely timestamped traces) to monitor $\rho_{\overline{\sigma}}$ with respect to $tf(\varphi)$.

They aim to identify the MTL fragments φ for which conformance with $tf(\varphi)$ over $\rho_{\overline{\sigma}}$ implies conformance of all $\pi \in TL(\overline{\sigma})$ with φ, which consequently implies the satisfaction of φ over $\overline{\sigma}$. For example, if $\varphi = p$ and $tf(p)$ is satisfied at t, then p is satisfied at some t' within the interval $[t - \delta; t + \delta]$, and thus there is a possible timeline for which φ is satisfied at t. However, not all timelines satisfy φ at t. In this case, we cannot obtain guarantees about a precise verdict of whether $\overline{\sigma}$ satisfies φ, so we obtain non-conclusive verdict "?". In contrast, for $\neg\varphi = \neg p$, we have $tf(p)$ is not satisfied at t, then φ is not satisfied on the interval $[t - \delta; t + \delta]$ on $\rho_{\overline{\sigma}}$, then there is no possible timeline satisfying p at t. Hence, we can obtain guarantees that $\overline{\sigma}$ satisfies φ. As a conclusion, the fragments of the property that can be satisfied (resp. violated) at all time points in the interval $[t - \delta; t + \delta]$ and consequently by all (resp. none) of the timelines π and emit the verdict $\top$ (resp. $\bot$) are those in which atomic propositions occur only negatively.

2.3.2.4 BASIN *ET AL.* [29]: RUNTIME VERIFICATION OF TEMPORAL PROPERTIES OVER OUT-OF-ORDER DATA STREAMS

Type of uncertainty targeted Basin *et al.* present an approach for runtime verification of properties over a data stream whose events may arrive to the monitor out of order or may not arrive due to delays and losses (this represents case 8 of Section 2.2.4).

Type of events and their representation The monitor observes a prefix of a timed word with gaps due to arbitrary message delays. These gaps may be filled when more messages arrive to the monitor from time to time. The timed word is a sequence of letters, and each letter is of the form $\langle I, \sigma\rangle$ where I is a non-empty interval describing a time point in the timed word and σ is a partial function describing an action. Initially the monitor does not know anything about the system behavior, so the timed word is represented as an infinite gap $\langle [0, \infty), []\rangle$. If a message (such as *"predicate p is true"*) arrives at timestamp 1, the interval $[0, \infty)$ will be split and the timed word becomes $\langle [0, 1), []\rangle\langle \{1\}, [p \to true]\rangle\langle (1, \infty), []\rangle$, and so on. If the monitor concludes that no action in the interval $[0, 1)$, the letter $\langle [0, 1), []\rangle$ can be removed and the timed word becomes $\langle \{1\}, [p \to true]\rangle\langle (1, \infty), []\rangle$.

Method used to represent specification property and the verdict type Basin *et al.* extend MTL (§1.4) into MTL$^{\downarrow}$ to reason about data values in the trace, where a freeze quantifier $\downarrow$ is used to take a value from a register in the state at a time point and freezes it into a variable. A freeze quantifier is a weak form of existential quantification. An MTL$^{\downarrow}$ policy example is:

$$\square \downarrow^{cid} c.\ \downarrow^{tid} t.\ \downarrow^{amt} a.trans(c, t, a) \wedge a \geq 2000 \rightarrow \\ \square_{(0,3]} \downarrow^{tid} t'.\ \downarrow^{amt} a'.\neg trans(c, t', a')$$

which states that "if a customer executes a transaction that exceeds \$2,000, then he must not execute any other transaction for 3 days". The registers cid, tid and amt stores the customer id, transaction id and the transferred sum respectively. The variables c and t are frozen to the values in cid and tid respectively. The variables a and a' are frozen to values stored in the register amt but at different times. The same for t and t'.

Basin *et al.* interpret the truth values as in Kleene logic and conservatively extend the logic's standard Boolean semantics as in [27]. MTL$^{\downarrow}$'s three-valued semantics is defined by $[[w, i, v \models \varphi]] \in \mathbb{B}_3$ where w is the observation, $i \in \mathbb{N}$ is the time point and $v : V \to D$ is a partial valuation that maps each logical variable to its value (V is the set of variables and D is the data domain). If $\varphi = t$ or $\varphi = f$, a precise verdict is simply produced. However, if $\varphi = p(\bar{x})$, then a precise verdict is produced only if $v(\bar{x})$ is defined, otherwise a non-conclusive verdict $\perp$ is emitted. If we have $\downarrow^r x.\varphi$, then the valuation v is obtained by freezing the value of x to the value in the register r. For the Boolean connectives $\neg$, $\vee$ and $\wedge$, the interpretation is trivial. However for other connectives such as $\mathbf{U}_I$, more interpretation is needed.

2.3.2.5 FERRANDO *ET AL.* [77]: RUNTIME VERIFICATION WITH IMPERFECT INFORMATION THROUGH INDISTINGUISHABILITY RELATIONS

Type of uncertainty targeted According to Ferrando *et al.*, the standard RV of LTL properties is based upon the assumption that the absence of an event a is considered equivalent to its negation $\neg a$, which is not true in a case where a exists but it is indistinguishable from another event. So, they focus on differentiating between knowing when something is not true and knowing when something is unknown.

Type of events and their representation Events are atomic propositions from an alphabet Σ. The absence of information is characterized by duplicating Σ such that $\bar{\Sigma} = \{p_\top, p_\bot, \forall p \in \Sigma\}$. The imperfect in information happens when atomic propositions such as p and q cannot be distinguished from each other (this represents case 2 od Section 2.2.4). This allows to introduce the equivalence relation $p \sim q$, the equivalent class $\gamma = \{p, q\}$, and the witness $[\gamma]_\top = \{p_\top, q_\top\}$ and $[\gamma]_\bot = \{p_\bot, q_\bot\}$.

They define two versions of traces: the *explicit* version σ_e where $p_\top \in \sigma_e(i)$ if p holds at $\sigma(i)$ and $p_\bot \in \sigma_e(i)$ if p does not hold at $\sigma(i)$; and the *visible* version σ_v derived from the σ_e where $[\gamma]_\top$ (resp. $[\gamma]_\top$) $\in \sigma_v(i)$ if $\forall p \in \gamma, p_\top$ (resp. $p_\bot$) $\in \sigma_e(i)$.

Method used to represent specification property and the verdict type Ferrando *et al.* use LTL to express the property φ. They define an explicit version $\epsilon(\varphi)$ of φ, where $\epsilon(p) = [\gamma]_\top$ and $\epsilon(\neg p) = [\gamma]_\bot$. They also define the operators $\vee$ and $\wedge$, as well as the *next* operator $\circ$, where $\epsilon(\circ\varphi) = \circ\epsilon(\varphi)$.

Ferrando *et al.* extend the standard monitor's synthesis pipeline (Figure 1.6 of Section 1.2) to explicitly consider imperfect information. They generate the DFA of $\epsilon(\varphi)$ to recognize the prefixes of trace that satisfy φ and $\epsilon(\neg\varphi)$ to recognize those that violate φ. However, the duplication of the atomic propositions in the formula breaks the duality between φ and $\neg\varphi$. For this reason, they added $\otimes\varphi$ which is $\neg\epsilon(\varphi) \wedge \neg\epsilon(\neg\varphi)$ and followed the same steps to generate the DFA of $\otimes\varphi$ which can recognize the prefixes having continuations that do not satisfy nor violate φ.

Each of the three monitors will process a visible trace δ_v and return a verdict in $\{\top, \bot, ?\}$. The resulting the three verdicts $v_{\epsilon(\varphi)}, v_{\epsilon(\neg\varphi)}, v_{\otimes\varphi}$ emitted by the DFA of $\epsilon(\varphi)$, DFA of $\epsilon(\neg\varphi)$ and DFA of $\otimes\varphi$ respectively, can be combined to deduce one final outcome. Five possible combinations exist: $\top$ if there is no continuation of δ_v which either violates φ or makes it undefined (i.e. $v_{\epsilon(\neg\varphi)} = \top, v_{\otimes\varphi} = \bot, v_{\epsilon(\varphi)} = \bot$). The verdict $\bot$ if there is no continuation which either satisfies φ or makes it undefined (i.e. $v_{\epsilon(\neg\varphi)} = \bot, v_{\otimes\varphi} = \top, v_{\epsilon(\varphi)} = \bot$). The verdict *uu* if there is no continuation which either satisfies or violates $\epsilon(\varphi)$ (i.e. $v_{\epsilon(\neg\varphi)} = \bot, v_{\otimes\varphi} = \bot, v_{\epsilon(\varphi)} = \top$). The verdict $?_{\not\bot}$ if there is no continuation which will eventually violate $\epsilon(\varphi)$, but there are continuations that satisfy $\epsilon(\varphi)$ and make it undefined (i.e. $v_{\epsilon(\neg\varphi)} = \top, v_{\otimes\varphi} = \bot, v_{\epsilon(\varphi)} = \top$). Symmetrically, the verdict $?_{\not\top}$ if there are no continuations satisfying $\epsilon(\varphi)$, but continuations that violate $\epsilon(\varphi)$ and make it undefined exist (i.e. $v_{\epsilon(\neg\varphi)} = \bot, v_{\otimes\varphi} = \top, v_{\epsilon(\varphi)} = \top$). Finally, the verdict "?" if the monitor cannot conclude anything yet, because there exist continuations satisfying $\epsilon(\varphi)$, continuations violating $\epsilon(\varphi)$, and continuations that make it undefined (i.e. $v_{\epsilon(\neg\varphi)} = \top, v_{\otimes\varphi} = \top, v_{\epsilon(\varphi)} = \top$).

2.3.2.6 ACETO *ET AL.* [4]: MONITORING FOR SILENT ACTIONS

Type of uncertainty targeted The approach conducted by Aceto *et al.* centers around the monitorability of a system that encounters *silent actions* or events. These actions refer to computational steps that are not revealed in the system model's level of abstraction. Nonetheless, the model presents sufficient indications of their occurrence throughout execution.

Type of events and their representation Two types of actions are represented using atomic symbols: external or observable actions and silent actions. The system states or processes are modeled as a standard labeled-transition system (LTS) model L, where actions stimulate the transitions between states. The processing of silent action is represented by a τ-transition. Several silent actions can happen successively causing a sequence of τ-transitions which can be obscured by turning them into υ-transitions, thus hiding how many transitions were taking place at certain points and obtaining an obscured LTS L' (the obscuring of the number of transitions is equivalent to case 3 of Section 2.2.4). Any state having a τ-transition in L still have a τ-transition in L'. External transitions are not affected and if a state p has a sequence of τ-transitions in L leading to a state q that can perform an external action, this observation is preserved in L'.

Method used to represent specification property and the verdict type Specification properties are expressed in a variant of the modal μ-calculus called μHML formulae (Hennessy Milner Logic) described in [121]. μ-HML is a dynamic logic with structure similar to an automaton and modal operators that also describe τ-transitions. Its operators include $true, false, \vee, \wedge, [\mu]\varphi, p$ which states that $\forall$ state q of the LTS reached by event μ from p, φ holds at q, and $\langle\mu\rangle\varphi, p$ which states that $\exists q$ reached by event μ from p and φ holds at q.

The monitoring setup is composed of an LTS system L and a monitor M consists of a set of states S_M and accepts external and silent actions. When the system produces a trace event μ that the monitor is able to analyze by transitioning from m to n, where $m, n \in S_M$, the constituent components of a monitored system $m \triangleleft p$ move in lockstep, where $m \triangleleft p$ means that the LTS is in state p when M is in state m. On the other hand, if M is unable to analyze an event μ, the monitored system still executes, but the monitor transitions to an *inconclusive state end*, where it remains for the rest of the computation.

Aceto *et al.* focus on rejection monitors to monitor *safety* fragments of the μHML formula, and use a state *no* to designate the rejection state. A monitor at state m rejects a process p in L if there exists a process q in L and a sequence of actions s such that the monitor ends in state *no* and the system is at state q after processing the trace s.

2.3.3 STATISTICAL-BASED SOLUTIONS

Another way to verify a property against a trace that contains uncertainty is to statistically compute the probability that the property is satisfied. In other words, the probability that a positive verdict is emitted.

2.3.3.1 STOLLER *ET AL.*: RUNTIME VERIFICATION WITH STATE ESTIMATION (RVSE)

Type of uncertainty targeted Stoller *et al.* [167] account for missing events in a trace and present the RVSE algorithm which is based on a statistical model of the monitored system. The aim is to fill the gaps and predict the probability that a positive verdict is emitted when encountering a gap.

Type of events and their representation Simple atomic symbols are used to represent the internal states of the system. A Hidden Markov Model (HMM) is used to represent the actual internal states of the system and can be learned from complete system traces using machine learning algorithms [81, 127, 156, 175]. The presence of a gap is represented by the symbol $gap(L)$, where L is a probability distribution representing the length of the gap and $L(l)$ is the probability that the gap has length l (this represents case 3 of Section 2.2.4).

Using the forward algorithm, at each time point t, the system is in some internal (hidden) state s_i, it undergoes a change to a next state s_k according to a transition probability, and emits an observation symbol o_j according to an observation probability. The result is a sequence of observation symbols such as $O = O_1, O_2, ..., O_t$.

Method used to represent specification property and the verdict type Stoller *et al.* use a DFA $\mathcal{M}$ to represent the property φ to be monitored. For each observation symbol emitted by the HMM, $\mathcal{M}$ moves from the current state to the next one. The sequence O satisfies the property iff it leaves the monitor $\mathcal{M}$ in an accepting state m_f when processing the last observation symbol O_t, and the probability of satisfaction is computed based on the transition and observation probabilities taking into account all ways of reaching the configuration in which the HMM is in state s_t and $\mathcal{M}$ is in state m_f. If a gap appears in the observation sequence when $\mathcal{M}$ is at state n and $\mathcal{H}$ is at state s_i, the observation symbol, say v, emitted by s_i cannot be determined. In this case, the extended forward algorithm sums over all the possibilities that the monitor can move from a predecessor p to n by adding the probability of each observation symbol between p and n.

Another statistical-based approach is proposed by Zhou *et al.* [191]. They first learn an HMM and transform it to a Discrete Time Markov Chain (DTMC), which is a stochastic process in which the next state depends only on the current state, and not any historical states. However, instead of using the classical forward algorithm, they used Baum-Welch algorithm (reader can refer to [2] for more details about the algorithm) to model the system based on the previously observed target event sequence.

2.3.3.2 KALAJDZIC *ET AL.* [106]: RUNTIME VERIFICATION WITH PARTICLE FILTERING (RVPF)

Type of uncertainty targeted The approach of Kaladjzic *et al.* is based on that of Stoller's *et al.* They also account for the presence of gaps in the trace. However, they introduce a technique for controlling the trade-off between runtime overhead and prediction accuracy.

Type of events and their representation Similar to Stoller *et al.*, the system states are represented using an HMM and each state emits an observation symbol. The symbol $gap(L)$ is used to denote a possible gap whose length is drawn from a probability distribution L over the natural numbers (this represents case 3 of Section 2.2.4). However, Kalajdzic *et al.* introduce a new type of events called *"peek events"*, which represent observations of parts of the program state, which are performed probabilistically at the end of a gap. Peek events help correct the movement errors introduced by using the HMM model during gaps. After each gap, a peek operation inspects a variable or a set of variables in the program state and returns an observation q_t. This information provided by a peek event helps reducing the uncertainty in the monitor state after gaps, which in turn narrows down the monitor DFA's possible states.

Method used to monitor and fill gaps and the verdict type Similar to Stoller *et al.*, the goal is to calculate a probability that the system's behavior satisfies φ, i.e. to produce a probabilistic verdict. However, in contrast to Stoller *et al.*, Kalajdzic *et al.* model the composition of the HMM and DFA as a Dynamic Bayesian Network (DBN), which is a type of Bayesian network that relates the system state variables x_t and the monitor state variables s_t to each other and to the observation variables o_t as well as to their previous states x_{x-1} and s_{t-1} over adjacent time steps. If a gap is encountered at time t, a peek event q_t is produced at the end of the gap.

Kalajdzic *et al.* proposed the RVPF algorithm where the system state is represented by a set of particles. A particle is a hypothetical state of the system being modeled which represents a possible value or configuration of the system's state, and is often drawn from a probability distribution that reflects the uncertainty in the state estimation. The idea is to represent the system state with a large number of particles and use them to estimate the probability distribution of the true state of the system. Particle filtering (PF) is used with sequential importance resampling (SIR) to estimate the internal state of the DBN. The importance weight of each particle in a state is summed to estimate the probability of that state. When an observed event occurs, each particle selects a state transition to execute by sampling the joint transition probability distribution of the DBN. The particles are then redistributed among the states that provided the best prediction of the current observation. By utilizing the DBN structure and the current observation, SIR is used to decrease the variance of the PF and enhance its performance.

2.3.3.3 WILCOX *ET AL.* [185]: RUNTIME VERIFICATION OF STOCHASTIC, FAULTY SYSTEMS

Type of uncertainty targeted Wilcox *et al.*'s monitor a safety property over mixed stochastic systems (which consist of both hardware and software components) that may suffer from state uncertainty as they degrade due to hardware failure, and imprecise or unobserved future states due to the possible interactions of their components (a state could be missing or uncertain but treated as missing, so this refers to cases 1 and 2 of Section 2.2.4).

Method used to represent specification property A safety constraint φ is written in LTL (Section 1.4) which is converted into automata, mainly an NBA to automate the monitoring. However, NBA does not guarantee a complete transition function of the safety requirement. Hence, an NBA is converted into a deterministic BA.

Type of events and their representation and the monitoring method The embedded system states are represented using Probabilistic Hierarchical Constraint Automata (PHCA) formalism. PHCA is similar to an HMM in the sense that it employs hidden states and probabilistic transitions. However, PHCA incorporates state constraints and a hierarchy of component automata. The system is modeled as a collection of individual PHCA components that communicate through shared variables. Each component is defined by discrete modes of operation, which represent both normal and faulty behavior. These modes can transition probabilistically or based on system commands, and can also be constrained by the modes of other components.

The states of the PHCA are: q_t: the safety state of the system at time t, defined as the state of the DBA that describes the safety constraint φ, x_t: system state at time t, c_t: system command at time t and z_t the observation at time t. If x_t is observable, q_t can be easily calculated from available information. Else, q_t cannot be known. However, one can estimate the probability that the system remains safe with φ by determining the probability distribution of the DBA state q_t, which is based on the history of observations $(z_{1:t})$ and commands $(c_{1:t})$. This probability distribution is called a belief state $B(y_t) = \sum_{x_t} P(q_t, x_t | z_{1:t}, c_{1:t})$, where $y_t = q_t \otimes x_t$.

The computation of the belief state over the BA is similar to the standard Forward algorithm for HMM belief state update. The subsequent state is predicted in a stochastic manner, taking into account the previous belief and transition probabilities of the models. This prediction is subsequently adjusted based on received observations. The observation probability $(P(z_t|x_t))$ and transition probability $(P(x_t|x_{t-1}, c_t))$ are both reliant on the physical system's model. In the case of an HMM, these probabilities are defined as a component of the system's model. However, for PHCA, these probabilities are determined by calculating the transition and observation probabilities of the specified components throughout the system.

2.4 SYNTHESIS OF THE EXISTING APPROACHES

In Section 2.3, various approaches that address the challenge of runtime verification when uncertainty exists in the underlying trace are described. However, these approaches vary in terms of the types of uncertainty considered, the formalism used to represent events, the specification languages for property representation, the monitoring methods/algorithms employed, and the types of verdicts produced. In this section, we aim to analyze and compare these differences. Furthermore, we evaluate each approach based on key features that characterize runtime verification, such as soundness, completeness, and monotonicity.

It is important to mention that our own approach is briefly included in this comparison for the sake of completeness. However, the specific details will be discussed in Chapter 3 and the results in Chapter 5.

2.4.1 COMPARISON BASED ON UNCERTAINTY REPRESENTATION

Approaches in Section 2.3 account for different types of uncertainty in the trace: missing events whose content is unknown, imprecise events whose content is not completely defined, events with imprecise timestamps whose time of occurrence is not clear, or unordered events that arrive to the monitor in an unknown sequence. Table 2.1 specifies the different types of events uncertainty for each approach and how each approach represents uncertain events.

Wang *et al.* (§2.3.1.2) and Basin *et al.* (§2.3.2.4) both account for unordered events. Wang *et al.* replace the whole trace (which is an abstract trace) by the set of all possible sequences or all possible orderings of the events, whereas Basin *et al.* used the timed word $\langle [0, \infty), [\ ] \rangle$ representing an infinite gap. Whenever a new event arrives, the timed word is split to insert the event at a specific time point. Leucker *et al.* (§2.3.1.1) account for three types of uncertainty: a missing event at time point t represented as $\perp t$, imprecise event at t is represented as $\top$ and

Approach	**Missing events**	**Imprecise events**	**Imprecise timestamps**	**Unordered events**	**Representation**
Our Approach	✓	✓			All possible valuations
Leucker (§2.3.1.1)	✓	✓	✓		$\bot$t or $\top$t or $\smile$
Wang (§2.3.1.2)	✓			✓	All possible sequences
Joshi (§2.3.2.1)	✓				χ
Basin (§2.3.2.2)	✓	✓			"?"
Basin (§2.3.2.3)			✓		$[t-\delta; t+\delta]$
Basin (§2.3.2.4)				✓	$\langle$ time interval , [ ]$\rangle$
Ferrando (§2.3.2.5)		✓			p and $\neg p, \ \forall p \in \Sigma$
Aceto (§2.3.2.6)	✓				ν-transition
Stoller (§2.3.3.1)	✓				$gap(L)$
Kalajdzic (§2.3.3.2)	✓				$gap(L)$
Wilcox (§2.3.3.3)	✓				Unobservable state x

Table 2.1 : Different methods to represent uncertainty in events.

a gap (a segment of the abstract stream representing all combinations of events in terms of timestamps and values) is represented as $\smile$. Basin *et al.* (§2.3.2.3) also account for imprecise timestamps in a trace represented as a timed word. They assume a timestamp imprecision $\delta \geq 0$ and an imprecise timestamp is assumed to belong to $[\tau_i - \delta, \tau_i + \delta]$. Based on this, several timelines are obtained from one timed word. In this thesis, we will replace a missing event with all possible valuations which means that each imprecise or missing event will be replaced by a set of possible replacements.

Joshi (§2.3.2.1), Stoller (§2.3.3.1) and Kalajdzic (§2.3.3.2) only account for complete loss of events, and not uncertain events. They simply represent a missing event by a symbol. Joshi *et al.* use the symbol χ, whereas Stoller *et al.* and kalajdzic *et al.* use the symbol $gap(L)$. Ferrando (§2.3.2.5) accounts for imprecise or indistinguishable events. If an event p is indistinguishable, it is represented as p and $\neg p$. For Wilcox *et al.* (§2.3.3.3), a missing event is simply an unobserved state x in the system model. As to Aceto *et al.* (§2.3.2.6), their approach is limited to one kind of uncertainty which is replacing a group of τ silent actions with one less precise υ silent action.

Basin *et al.* (§2.3.2.2) use a model that represents uncertainties over event occurrences. A knowledge gap with regard to whether a predicate $r(a_1, \ldots, a_n)$ happened at τ is represented as $r(a_1, \ldots, a_n) =?$. In the case of Basin (§2.3.2.3), they represent an event as a tuple where each timestamp is replaced by an interval of time.

2.4.2 COMPARISON BASED ON VERDICT REPRESENTATION

Runtime verification is all about producing one precise verdict for each event. However, the presence of uncertain or missing events makes this challenging due to the inability of the monitor to precisely observe the event and correctly emit a verdict. Changing the format of representing events in a trace to account for the imprecision in their content may change the form (in terms of structure) and the type (in terms of number) of the output verdicts. Table 2.2 shows for each approach in Section 2.3, the type of the produced verdict and the form used to represent it.

Some approaches produce a set of verdicts instead of one verdict. In our approach, we will replace each input event by the set of possible valuations and emit a verdict for each valuation. The verdict produced will take the form of a set of verdicts rather than one single verdict. We will also quantify each verdict by counting how many uni-traces projections of a multi-trace result in each verdict, producing a form of “probability” or “likelihood”. Similarly for Leucker (§2.3.1.1), whose approach emits a verdict for each concrete event, resulting in a set of verdicts (abstract verdict) for an abstract event. The result is an abstract verdict representing a set of the verdicts of the concrete events.

Stoller (§2.3.3.1), Kalajdzic (§2.3.3.2) and Wilcox (§2.3.3.3) use a probabilistic model such as HMM and PHCA to represent the system states, and estimate their verdicts as a probability that an event satisfies the property. Other approaches rely on an extension of existing formalisms to define new verdicts that account for missing/uncertain events, and produce a single verdict, such as Joshi (§2.3.2.1) who extends the existing RV-LTL to produce the non-conclusive verdict "?" when processing the symbol χ. Similarly, Ferrando (§2.3.2.5) extend the standard monitor's pipeline to include the verdicts $\{uu, ?_{\not\perp}, ?_{\not\top}, ?\}$ and explicitly consider imprecise events. Aceto (§2.3.2.6) focus on rejection monitors for safety fragments of their policy, which ends in state *no* if a process is rejected and in a non-conclusive state *end* if the event cannot be analyzed (meaning that the property is non-monitorable). Their approach is limited in that it only accounts for violations. However, it tackles issues related to monitorability which are not considered in the other approaches discussed in this survey.

The rest of the approaches in Table 2.2 produce a single verdict in $\mathbf{B}_3$ such as Basin *et al.* (§2.3.2.2, §2.3.2.3, §2.3.2.4) and Wang *et al.* (§2.3.1.2). Wang *et al.* generate all the possible sequences (concrete traces) consistent with the given abstract trace and produce a concrete verdict for each concrete trace. However, the final verdict is a single verdict $\top$ if all the concrete verdicts are $\top$, $\perp$ if all concrete verdicts are $\perp$, or "?" otherwise.

In scenarios involving uncertain or missing events, guaranteeing soundness, completeness and monotonicity is challenging because the presence of knowledge gaps could limit the ability of the monitor to detect some violations or satisfactions of the specified properties, which affects the soundness and completeness of the monitor. On the other hand, one must ensure that the verdicts persist after closing knowledge gaps to guarantee the monotonicity. Table 2.3 states for each of the approaches of Section 2.3 whether it guarantees soundness, completeness and monotonicity.

Approach	One-verdict	Multi-verdicts	Probability	Form
Our Approach		✓		Set of verdicts from $\{\top, \bot, ?\}$
Leucker (§2.3.1.1)		✓		Set of verdicts from $\{\top, \bot, ?\}$
Wang (§2.3.1.2)	✓			One value from $\{\top, \bot, ?\}$
Joshi (§2.3.2.1)	✓			One value from $\{\top, \top_p, ?, \bot_p, \bot\}$
Basin (§2.3.2.2)	✓			One value from $\{\top, \bot, ?\}$
Basin (§2.3.2.3)	✓			One value from $\{\top, \bot, ?\}$
Basin (§2.3.2.4)	✓			One value from {t, f, $\bot$}
Ferrando (§2.3.2.5)	✓			One value from $\{\top, \bot, uu, ?_\bot, ?_\top, ?\}$
Aceto (§2.3.2.6)	✓			One value from $\{no, end\}$
Stoller (§2.3.3.1)			✓	Probability of satisfaction
Kalajdzic (§2.3.3.2)			✓	Probability of satisfaction
Wilcox (§2.3.3.3)			✓	Probability of satisfaction

Table 2.2 : Different methods to represent verdicts.

The statistical-based approaches (Stoller *et al.* (§2.3.3.1), Kalajdzic *et al.* (§2.3.3.2), Wilcox *et al.* (§2.3.3.3)), use a model of the system (an HMM or a PHCA) to estimate the probabilities of hidden states, fill the gaps and generate a sequence of observation symbols representing the most likely sequence of hidden states and emitted events. However, it is important to note that the accuracy of the imputed events depends on the accuracy of the HMM and the estimated probabilities. If the HMM does not accurately capture the underlying system behavior or the estimated probabilities are unreliable, the generated sequence may not accurately reflect the actual system behavior, leading to false positives (incorrectly reporting a violation) and false negatives (failing to report a violation). Therefore, guaranteeing soundness and completeness of the statistical-based approaches is challenging. The same can be with respect to the monotonicity, as the verdict's consistency cannot be guaranteed.

In our work, we assume that all valuations of the input multi-event are still valuations of the output multi-event, hence the verdicts that are supposed to be emitted for the input valuations will be preserved. Consequently, *False* verdicts are still emitted for violating events, and *True* verdicts are still emitted for valid events, which guarantees the soundness of the approach. With respect to monotonicity, as each missing event will be replaced by the set of all possible replacements and the multi-verdict will include all the possible verdicts, this means the same multi-verdict will be produced for any replacement. Hence verdicts are also preserved. Similarly, the approach of Leucker *et al.* (§2.3.1.1) states that the verdict of each concrete trace persists in the abstract trace. This feature guarantees the soundness and monotonicity of the approach. The approach of Wang *et al.* (§2.3.1.2) states that a true (resp. false) verdict is emitted when monitoring a property over an abstract trace if and only if all the concrete traces consistent with this abstract trace evaluates to true (resp. false). Otherwise, the outcome is uncertain. This guarantees the monotonicity of the approach because the same verdict will be produced for any possible replacement of events. The approach is also sound because it produces correct verdicts. However, our approach as well as that of Leucker *et al.* §2.3.1.1 and Wang *et al.* §2.3.1.2 are not complete because, in some cases, an uncertain verdict is produced.

With respect to Joshi *et al.* (§2.3.2.1), the loss-tolerant monitor $\mathcal{M}$ guarantees soundness because it produces a verdict at the end of the trace compatible with that of an RV-LTL monitor, assuming that a loss tolerant cluster and a loss tolerant alphabet exist. However, some patterns such as $\square(a \rightarrow (b \wedge c))$ cannot be soundly monitored under transient loss. The approach guarantees monotonicity because, as described by Joshi *et al.*, the output of the $\mathcal{M}$ is always equal to that of an RV-LTL (before and after processing the lossy elements χ). Since the outputs of an RV-LTL monitor is monotonic, we conclude that the output of $\mathcal{M}$ is also monotonic. However, the approach is not complete since the state machine produces the uncertain verdict "?" when processing χ.

The approach of Basin *et al.* (§2.3.2.2) aims to avoid reporting a policy violation unless there is indeed a violation. This implies that the approach is sound. Their approach is not complete in the sense that some policy violations may not be reported. However, completeness is guaranteed on an expressive fragment of the compliance policy that retains all the language's connectives but limits the usage of free variables. With respect to the monotonicity requirement, the policy language used by Basin *et al.*'s work ensures that this requirement is maintained. In this language, evaluations of formulas do not reduce the amount of knowledge when resolving incompleteness in the extension of a logging knowledge base.

The approach of Basin *et al.* (§2.3.2.4) provides soundness and completeness guarantees in the sense that verdicts are correct w.r.t. the observations given to the monitor, meaning that, assuming no failures occur, violations and satisfactions of specifications will eventually be reported despite the presence of finite message delays. Their reasoning is monotonic with respect to the partial order on truth values, where $\perp$ is less than t and f, and t and f are incomparable. This monotonicity property ensures that closing knowledge gaps does not contradict previously obtained Boolean truth values. In other words, when filling a knowledge gap represented by $\perp$ with either t or f, the resulting truth value will always be consistent with the previously obtained one.

The approach of Basin *et al.* (§2.3.2.3) is sound in the sense it always emits a correct verdict. However, soundness is guaranteed only for certain MTL fragments in which atomic propositions occur only negatively. The approach is also complete for these fragments because the same precise verdict is emitted for all the timelines and for the timed word $\overline{\sigma}$. Similar to Basin *et al.* (§2.3.2.4), the approach is monotonic.

The approach of Ferrando *et al.* (§2.3.2.5) is sound in the sense that all the emitted verdicts are correct. In other words, a negative verdict is emitted only if a violation occurs and a positive verdict is emitted only if a satisfaction happens. However, the algorithm is not complete in the sense that at some point, no verdict is emitted (represented by "?") which means that a satisfaction or a violation is missed. Monotonicity is guaranteed because the

Approach	Monotonic	Complete	Sound
Our Approach	✓	✗	✓
Leucker (§2.3.1.1)	✓	✗	✓
Wang (§2.3.1.2)	✓	✗	✓
Joshi (§2.3.2.1)	✓	✗	✓
Basin (§2.3.2.2)	✓	✓	✓
Basin (§2.3.2.3)	✓	✓	✓
Basin (§2.3.2.4)	✓	✓	✓
Ferrando (§2.3.2.5)	✓	✗	✓
Aceto (§2.3.2.6)	✓	✓	✓
Stoller (§2.3.3.1)	U	U	U
Kalajdzic (§2.3.3.2)	U	U	U
Wilcox (§2.3.3.3)	U	U	U

Table 2.3 : Features and limitations in related works (U = "Undetermined").

verdict $\top$ is produced if and only if there is no continuation of δ_v which either violates φ or makes it undefined, and the verdict $\perp$ is produced if and only if there is no continuation which either satisfies φ or makes it undefined. This means that once the verdict $\top$ or $\perp$ is emitted, it persists over all the possible continuations of the trace.

According to Aceto *et al.* (§2.3.2.6), their monitor can check for a μHML formula φ on L from any obscuring L' of L if $\forall p$ in L': p does not satisfy φ on L iff p is rejected by the monitor on L'. So, the verdict produced when monitoring over L' is compatible with the verdict produced when monitoring over L. Hence, the approach is guaranteed to be sound. However, similar to Basin *et al.* (§2.3.2.2), completeness is guaranteed only for a fragment of the μHML formula. Aceto *et al.* state that once the monitor transitions to the inconclusive state *end* (resp. rejection state *no*), it remains in this state for the rest of the computation. This indicates that the approach is monotonic.

Approach	Event Type	Policy
Our Approach	Valuation over Boolean variables	DFA
Leucker (§2.3.1.1)	Timestamp, data value	TeSSLa
Wang (§2.3.1.2)	Atomic event or process variable	Past-LTL
Joshi (§2.3.2.1)	Atomic symbol	LTL
Basin (§2.3.2.2)	Predicate with arity	FOTL
Basin (§2.3.2.3)	Tuple (atomic symbol, timestamp)	MTL
Basin (§2.3.2.4)	Tuple (time interval, atomic symbol)	Freeze MTL
Ferrando (§2.3.2.5)	Atomic symbol	LTL
Aceto (§2.3.2.6)	Atomic symbol	μHML Logic
Stoller (§2.3.3.1)	Observation symbol	DFA
Kalajdzic (§2.3.3.2)	Observation symbol	DFA
Wilcox (§2.3.3.3)	Observation symbol	LTL

Table 2.4 : Different methods to represent events and policies.

2.4.3 COMPARISON BASED ON SPECIFICATION LANGUAGE

The approaches in Section 2.3 use various specification languages to express the specification property. The languages are summarized in Table 2.4. Each specification language is characterized by its operators and expressiveness. Some approaches such as that of Joshi *et al.* (§2.3.2.1), Ferrando *et al.* (§2.3.2.5) and Wilcox *et al.* (§2.3.3.3) simply use the LTL formalism to express properties using atomic propositions and Boolean connectives. To automate the monitoring process Joshi *et al.*, Ferrando *et al.* and Wilcox *et al.* convert the LTL into FSM, DFA and BA respectively to automate the monitoring process.

Others such as our approach and that of Stoller *et al.* (§2.3.3.1), Kalajdzic *et al.* (§2.3.3.2) directly use finite state machines to represent the property. Aceto *et al.* (§2.3.2.6) use μHML formulae (Hennessy Milner Logic) whose structure is similar to an automaton. Additionally, it has modal operators that describe τ-transitions.

Some approaches use an extension of LTL such as Wang *et al.* (§2.3.1.2) who use Past-LTL which augments the LTL with operators that reason about the past. While Past-LTL does not offer greater expressiveness than LTL, it is much more concise and convenient for defining correctness properties when it comes to runtime verification over finite traces [152]. Since the events in their approach lack explicit timestamps, only linear time properties in LTL are analyzed. Basin *et al.* (§2.3.2.2) also propose an augmented LTL specification language that use the operators of LTL and propose more connective operators to reason about the past and future time points and operators to reason about incompleteness and handle inconsistencies. Another language used by Basin *et al.* (§2.3.2.3) is MTL which extends LTL with timing constraints over the temporal operators to reason about the imprecision in timestamps. Later, Basin *et al.* (§2.3.2.4) extended the MTL with freeze variables to reason about data values in the trace. MTL and Freeze MTL have more operators than LTL and allow specifying time constraints. However, the presence of freeze quantifiers and temporal connectives in the specification property increases the running time of the monitoring algorithm.

The above logics are common in static verification and are not suitable for stream runtime verification. In contrast, Leucker *et al.* (§2.3.1.1) extended the existing TeSSLa specification language into Abstract TeSSLa and propose an abstract operator for each concrete operator of TeSSLa. Their language is suitable for monitoring streams and is equipped with operators to reason about imprecise timestamps which increases its expressiveness.

The relative expressiveness of these languages cannot be established in a clear-cut manner. It is known that propositional LTL, past LTL and DFA are equivalent for finite prefixes of a trace. The remaining specification languages are strictly more expressive than those three, although a strict ordering between them is not known.

2.4.4 COMPARISON BASED ON EVALUATION METHODS

A last element of comparison between these works is the empirical assessment of their performance. Table 2.5 summarizes the methods that each of the approaches in Section 2.3

rely on to evaluate their work and the results they obtained.

With respect to our approach, we aim to run tests across a variety of uncertainty scenarios to determine the overhead imposed by the existence of the access proxy and multi-monitor in terms of both running time and memory consumption. In terms of running time, we shall see in Chapter 5 that the presence of an access proxy (the model we use to represent uncertainty) causes a slowdown in the monitoring process because the monitor must handle multi-events rather than uni-events and track the many possible states of the uni-monitor. However, for the scenarios considered, this slowing shall not exceed 8×. In terms of memory consumption, having many events would increase the maximum amount of memory consumed by the monitor, but this increase is minor and never reaches a factor of 1.5.

Leucker *et al.* (§2.3.1.1) perform empirical evaluation on different TeSSLa specifications to evaluate the computational overhead in terms of how many concrete TeSSLa operators are needed to realize the Abstract TeSSLa specification. Results showed that evaluating the abstract specification typically only increases the computational cost by a constant factor, and if a concrete specification can be monitored in linear time (in the size of the trace) its abstract counterpart can be as well.

Wang *et al.* (§2.3.1.2) test a number of properties over an actual number of traces. The experiments show that 97.7% of the running time was spent on executing the *checkOne* function, due to the exponential number of concrete traces corresponding to an abstract state. With respect to the frequency of the resulting uncertain verdicts, the results show that a low number of traces (15.61%) end in inconclusive verdict. This is justified by the fact that most of the temporal operators are insensitive to the uncertainty, and also the scope of uncertainty is bounded within one abstract state.

Joshi *et al.* (§2.3.2.1) show that the additional overhead incurred by loss-tolerant monitor

Approach	Evaluation Factor	Result
Our Approach	Running time	less than 8×
	Memory consumption	Less than 1.5
Leucker (§2.3.1.1)	Computational cost	Increased by constant factor
Wang (§2.3.1.2)	Running time of an abstract state	97% of the total running time
	Frequency of uncertain verdicts	15.61%
Joshi (§2.3.2.1)	Memory consumption	Between 5 and 534 transitions
	Time complexity	$m \times n \times 2^n$
	Complexity	$O(N_h^2 \times N_d)$
Basin (§2.3.2.4)	Running time	Increases rapidly
Ferrando (§2.3.2.5)	Monitor execution time	Linear w.r.t. the trace length
	Monitor synthesis time	Exponential w.r.t. LTL length
Stoller (§2.3.3.1)	Inaccuracy	15× better than naive approach
	Time complexity without gap	$O(N_h^2 \times N_d)$
	Time complexity with gap	$O(N_h^2 \times N_d^2)$
Kalajdzic (§2.3.3.2)	Memory consumption	$16 \times N_p + 3560$
	Execution time	Outperforms RVSE of Stoller
Wilcox (§2.3.3.3)	Time complexity	$O(n^2)$
	Space complexity	$O(n)$

Table 2.5 : Empirical evaluations of different approaches.

$\mathcal{M}$ due to additional states is not significant (only an increase of at most two from that of the corresponding RV-LTL monitor). The overhead in terms of memory at runtime due to the increased number of transitions in $\mathcal{M}$ is also proved to be minimal (fluctuating between 5 and 534 extra transitions) compared to an RV-LTL. With respect to the time complexity of the monitoring algorithm, it is exponential with the number of states of $\mathcal{M}$ and is equivalent to $m \times n \times 2^n$ where m is the size of Σ and n is the number of state in $\mathcal{M}$. It is also proved that the size complexity of $\mathcal{M}$ is identical to that of the RV-LTL monitor.

Basin *et al.* (§2.3.2.4) perform experiments to evaluate the effect of freeze quantifiers and temporal connectives in the specification property on the running time. They also offset the arrival time of an event by a random delay to evaluate the effect of out-of-order arrival on the running time. The results show an increase in the running time for formulas containing more freeze quantifiers and temporal connectives. The running time also increases when messages are received out-of-order.

Ferrando *et al.* (§2.3.2.5) carried out experiments by varying the length of the trace of events. The results showed that the execution time is linear w.r.t. the length of the trace, then the time required for the monitor to analyze a single event in the trace is constant. They also measure the execution time for the monitor synthesis from LTL formulas with different lengths and proved that increases exponentially with the size of the input formula.

Stoller *et al.* (§2.3.3.1) measure the overall inaccuracy (i.e. how many events are not observed due to sampling), and obtained a ratio of 0.0205. This level of inaccuracy is quite low, considering the severity of the sampling. In comparison, the inaccuracy of a naive approach that ignores gaps due to sampling is 0.3135; this is approximately 15× worse. With respect to time complexity, it is $O(N_h^2 \times N_d)$ for a single observation without a gap event and $O(N_h^2 \times N_d^2)$ for a gap event, where N_h and N_d are the numbers of states of the HMM and the DFA, respectively.

Kalajdzic *et al.* (§2.3.3.2) conduct experiments to evaluate the effect of the number of particles N_p on execution time and memory consumption. The amount of required memory is a linear function of the number of particles and was measured to be $16 \times N_p + 3560$ using a 10-state HMM. Compared to RVSE, this is higher, and in comparing to the AP-RVSE, it is around 80 times lower. In terms of execution time, RVPF outperforms RVSE for all gap lengths with increasing number of particles.

Wilcox *et al.* (§2.3.3.3) The cost of computing that a state is safe is entirely dependent on the sizes of Q and X. In order to find the probability of each q_t, we must loop twice over these sets. If n is the size of the combined set, $n = |Q \times X|$, then we have a time complexity of $O(n^2)$, and a space complexity of $O(n)$.

With respect to the rest of the approaches in Table 2.5 (Basin *et al.* §2.3.2.2–2.3.2.3 and Aceto *et al.* §2.3.2.6), no empirical evaluations are provided.

2.4.5 POSSIBLE EXTENSIONS OF THE RELATED APPROACHES

Sections 2.3 and 2.4 provided a comprehensive representation and comparison of the existing literature on monitoring with incomplete traces. It discussed the various sources of uncertainty that have been identified and examines their impact on the monitoring process. The survey evaluated and compared different approaches for monitoring incomplete traces, taking into consideration the types of uncertainties addressed, representations of uncertain events, the formalism used for event and policy representation, and the methods of representing output verdicts. The advantages and limitations of each approach were also highlighted based on their respective evaluation results. The thorough analysis of the surveyed works allows us to identify several areas where future research is warranted. We list the main ones in the following.

The approach developed by Wang *et al.* (§2.3.1.2) can be expanded to incorporate the complete semantics of LTL, including past-time and future-time operators. However, this extension would require significant effort. Another potential extension could address imprecise timestamps in recorded traces. Indeed, in the recorded traces, abstract states are timestamped when the state is recorded, but the time of actual observations is lost, which introduce uncertainty for timed operators. Therefore, it would be valuable to explore techniques for handling imprecise timestamps and addressing the impact of uncertainty on real-time properties.

Joshi *et al.* (§2.3.2.1) research could be extended to allow the approach to handle timed traces and more complex system architectures, such as distributed and concurrent systems. The approaches of Basin *et al.* ((§2.3.2.2) and (§2.3.2.3)) could be enhanced by conducting case studies to evaluate their effectiveness in real-world settings. Similarly, Aceto *et al.* (§2.3.2.6,) could assess the performance of their proposed monitoring approach by proposing a monitoring algorithm and conducting experiments. Basin *et al.* (§2.3.2.2) would also explore different truth spaces to distinguish between different kinds of knowledge gaps and disagreements.

While the experimental evaluation of Basin *et al.*'s approach (§2.3.2.4) is promising, their method currently cannot handle the monitoring of systems generating thousands or millions of events per second. Further research is necessary, including algorithmic optimization, which the authors plan to undertake in the future, as well as deploying and evaluating their approach in large-scale case studies. Finally, future investigations could focus on the empirical assessment of the time and space complexity of the monitoring process. Ferrando *et al.* (§2.3.2.5) plan to expand their approach in future work by proposing a method to add additional information to the monitor's verdict. This method would utilize the event trace, the LTL property, and the monitor's verdict to establish a level of confidence in the final outcome. Specifically, in cases where the outcome is *uu*, instead of simply stating that the property is undefined with respect to the trace, they could use a probability distribution over the relevant equivalence classes to assert that the property would be satisfied (or violated) with a certain probability threshold.

In the approach of Stoller *et al.* (§2.3.3.1), the matrix-vector calculations performed by the RVSE algorithm to get the transition and observation probabilities when processing any observation symbol makes the computation cost very high and increase the overhead dramatically, especially in the presence of large gaps. One future direction is to tackle this problem. Bartocci *et al.* [24] propose *approximately precomputed* RVSE (AP-RVSE) that significantly reduces the runtime overhead of RVSE by precomputing and storing the results of the matrix calculations performed by RVSE. However, their approach introduces some approximation error. With respect to Kalajdzic *et al.* (§2.3.3.2), an interesting extension could involve creating a runtime-variable version of RVPF, in which the number of particles employed for state estimation can be adjusted dynamically. This would enable the flexible control of the tradeoff between estimation accuracy, memory consumption, and speed. The approach of Wilcox *et al.* (§2.3.3.3) has undergone preliminary validation, which demonstrates its ability to rapidly and precisely identify safety violations in small models. Their future efforts might focus on determining the effectiveness of these methods on larger models.

Our work will be the only work with an explicit modeling of noise, in the form of the proxy, which makes it possible to model various kinds of perturbations (or data degradation) on an input trace and observe their effect on the monitor. For example, a valuation can simply swap the assignments of events a and b to make them indistinguishable: an input multi-event that supports a is transformed into an output multi-event that only supports the weaker proposition $a \vee b$ (and similarly for events that support b). In other words, it is no longer possible to conclude precisely that a is true or that b is true, only that at least one of them is true. Ferrando *et al.* (§2.3.2.5) can represent this form of uncertainty as an equivalence relation $a \sim b$. Some language-based approaches do not account for uncertain events (e.g. §2.3.2.3, §2.3.2.4) and other approaches (§2.3.2.1, §2.3.2.2, §2.3.2.6) are limited in their ability to account for uncertainty. The best these approaches can do is to approximate uncertainty by assuming that the occurrence of both a and b is unknown. However, this abstraction is only precise for events where neither a nor b are true. When dealing with abstract data domains such as that of Leucker *et al.* (§2.3.1.1), the situation becomes even less desirable. These domains are defined for each variable separately and must remain consistent throughout the entire trace. Therefore, the only way to preserve the world when abstracting is to replace the values of a and b with all possible values at all time points, resulting in an even greater over-approximation. Statistical-based approaches (§2.3.3.1, §2.3.3.2 and §2.3.3.3) can be also extended to deal with imprecise events. An imprecise event $a \vee b$ can be treated in the same way as a missing event (gap), however, assuming the monitor is at state n, one can add only the transition probabilities of the predecessors of n where $a \vee b$ holds instead of summing over all the predecessors of n.

Each of the approaches surveyed should consider addressing the problem of incompleteness as an additional future extension. Table 2.3 shows that all of the approaches surveyed (except the statistical-based ones) guarantee soundness. However, most of them are not complete. Even those considered as complete (§2.3.2.2, §2.3.2.3, §2.3.2.4, §2.3.2.6) provide completeness only for some fragments of their specification policy.

An interesting future extension of the runtime verification approaches under uncertainty is to study the effect of uncertainty on the monitored property. A property could be robust with respect to the existing type of uncertainty, i.e. still produces the correct verdict despite the imperfect events in the trace. For example, the property "every a is eventually followed by c" is robust to a type of trace corruption where events b and d are indistinguishable. One could also modify the property to make it more robust. In the work of Alechina *et al.* [9] for example, instead of modifying the trace to enhance the observation capabilities, they show how to synthesize an approximation of an "ideal" norm that can be perfectly monitored given a monitor, and which is optimal in the sense that any other approximation would fail to detect at least as many violations of the ideal norm.

A runtime verification approach could be also improved by creating a specification formalism that provides explicit constructs to express constraints on the system's behavior that take into account the possibility of imprecise events directly from within the property. For example, a property can constrain imprecise events to correspond to at most n concrete events which help ensure that the system is able to maintain a desired level of accuracy or precision in its behavior. This would also minimize the number of possible replacement traces of the input trace and the number of output verdicts. The specification formalism could also specify that no trace should contain more than n successive missing events. By imposing this limit, the specification can ensure that the system is able to recover from errors or unexpected inputs. By including explicit constructs for reasoning about uncertainty and imprecision, the specification formalism may be able to provide more precise and flexible ways to specify requirements for runtime verification of complex systems. However, it is important to carefully design and validate such constructs to ensure that they are useful and effective in practice.

2.5 EXISTING APPROACHES ON RUNTIME ENFORCEMENT

Runtime enforcement [72, 150, 163] seeks to react to any observed violation in such a way as to correct and recover from it, for example by modifying the execution or skipping execution steps. Similar to RV, the execution of the target system is abstracted as a sequence of program events, termed the input sequence. The security policy is usually, but not necessarily, a predicate over individual sequences, in which case, the terms “policy” and “property” can be used interchangeably.

These security mechanisms have been applied successfully to the field of telecommunications, notably to ensure conformity with cryptographic protocols [30, 183], and other network protocols [18, 94] and to monitor client—server communications in online stores [164].

Of particular importance in this thesis is an Enforcement Model (EM), which is defined as a processing unit that can transform an input sequence of events into another sequence.

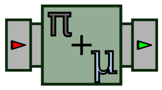


Figure 2.2 : A symbolic representation of an EM. Events enter on the left-hand side, and a corrected version of the input stream is produced on the right-hand side.

An EM is said to satisfy the *soundness* condition if its application on an input sequence always results in an output sequence that satisfies a given policy. An EM can be represented graphically as in Figure 2.2; symbols π and μ represent entities called the "proxy" and the "policy monitor", which we shall define in later sections. Depending on the type of monitor used, it may be subject to other constraints that limit the freedom of the monitor to substitute one sequence for another (a property we call *transparency*).

A long line of research focuses on delineating the set of properties that are (or are not) enforceable by monitors operating under a variety of constraints [33, 113, 163]. A key finding of these works is that the enforcement power of monitors is affected both by the capabilities of the monitor as an enforcement mechanism and by the licence given to the monitor to alter the input sequence (the transparency requirement). The design and capabilities of the proxy will naturally have a profound influence on the enforcement power of the complete pipeline [112]. A thorough survey of runtime enforcement, stressing its connection to runtime verification, is given by Falcone *et al.* [75].

2.5.1 MONITOR CAPABILITIES

In his initial formulation, Schneider [163] considered a monitor that observes the sequence of events produced by the target program and reacts by aborting the execution (truncating the execution sequence) upon encountering an event which, if appending that event to the ongoing execution, would violate the security policy. Truncation is the only remedial path available to such monitors, and the output of the monitor is thus necessarily the longest prefix of the input sequence that satisfies the desired property. The security automaton is limited to enforcing safety properties [95], which states that nothing bad happens during program execution. However, the enforcement power of the monitor can be extended if it has access to the results of a static analysis of its target's code. Such an analysis allows the monitor to build a model of the target program's possible behavior, enhancing the mechanism's enforcement power [46].

Ligatti *et al.* [33] consider more varied models of monitors that are capable of inserting events in the execution stream, of suppressing the occurrence of some events while allowing the remainder of the execution to proceed, or both. This categorization gives rise to a hierarchy of proxies with those that have edition capabilities being the most powerful. Suppression and insertion monitors have capabilities that are orthogonal to one another, and truncation monitors are the least powerful ones. Another characterization, in which some events lie beyond the control of the monitor, was proposed by Khoury *et al.* [110].

Extending the available capabilities given to a monitor to alter the input trace greatly extends its enforcement power but may also introduce several possible corrective courses of action to restore compliance with a policy. For instance, a trace where a *send* action occurs immediately after a file is being *read* violates a policy stipulating that no information can be sent on the network after reading from a secret file, unless the sending is recorded in a log beforehand. This is a slightly more involved version of a property already proposed by Schneider [163]. An execution violating this policy could be one in which the file is read, and a *send* action subsequently occurs, without an intervening *log* action. Multiple corrective actions are hence possible: aborting the execution before the *send* action (truncation); inserting an entry in the log (insertion); or suppressing either the *read* or the *send* action (suppression).

Later research distinguishes further subcategories of these monitors, derived from finer limitations on the ability of the monitor to alter the input sequence. For instance, the monitor may be limited to inserting events that have previously been suppressed [37, 38] or limited in its capacity to insert or suppress certain events [63, 110]. After all, the insertion or suppression of some events may be beyond the control of the monitor for a variety of reasons, such as computability constraints or because performing the underlying program actions requires encryption keys. The presence of uncontrollable actions brings a case-by-case subtlety to the question of enforceability.

In this line of research, the monitor is usually modeled as a finite-state machine, which dictates its behavior according to the input action and its current state. Care must be taken to ensure that this FSM correctly enforces the policy and is concordant with the limitations imposed on the monitor's capabilities. Falcone *et al.* [72] showed that a finer automaton model, with explicit store and dump operations, can enforce policies in the *response* class from the safety-progress classification [47]. Their model also lends itself to implementation in a more straightforward manner than previous models.

Another line of research examines how memory constraints affect the enforcement power of monitors. Thali *et al.* [172] study the enforcement power of monitors with bounded memory; Fong *et al.* [79] study a monitor that only records the shallow history (i.e. the unordered set of events) of the execution, while Beauquier *et al.* [34] study the enforcement power of a monitor with finite, but unbounded memory. On their side, the monitors proposed by Ligatti *et al.* and Bielova *et al.* have the capacity to store an unbounded quantity of program events, simulating the execution until it can ascertain that the ongoing execution is valid; however, this course of action may not always be possible in practice. In contrast, Dolzhenko *et al.* propose a model of monitoring in which the monitor is required to react to each action performed by the target program as it occurs [63].

2.5.2 TRANSPARENCY CONSTRAINTS

Several authors also considered how the licence given to the monitor to alter valid and invalid sequences affects its enforcement power, and in fact they have found that this aspect of monitoring is in some way even more consequential than the monitor's capabilities when delineating the set of properties that are enforceable by a monitor. In the original definition of runtime enforcement reported above, the notion of transparency only imposes that the monitor must maintain the semantics of *valid* sequences [33], which can lead to undesirable behavior. As an example, consider the policy "an opened file is eventually closed" and a sequence in which multiple files are consecutively opened and closed, except the final file, which is opened, but not closed. The monitor may correct the situation either by appending a close action at the end of the sequence or by deleting the opening of the ultimate file and any subsequent file actions (reads and writes). However, the monitor could also enforce the property by removing every well-formed pair of files being opened and closed, or even by adding to the sequence new events not present in the original. This is because the definition of enforcement entails that the monitor can replace an invalid sequence with *any* valid sequence, even one completely unrelated to the original execution.

Transparency constraints refer to mechanisms by which the available enforcement actions of a monitor are restricted according to some requirement. Indeed, when using the definition of enforcement given above, a monitor is said to enforce a property as long as it can replace an invalid sequence with *any* valid sequence, even one completely unrelated to the input the monitor has received. For example, Bielova *et al.* create sub-classes that further constraint the monitor's handling of invalid executions [37]. First is the class of monitors that is limited to delaying the execution of some program events, but may not insert new events into the execution; second, monitors that may only insert the delayed part of the execution on an all-or-nothing basis; third, monitors limited to output some prefix of invalid sequences. They compare the set of enforceable properties in each case. In the example given above, the transparency constraint could be based on the number of completed open-close pairs. This would prevent the monitor from deleting valid parts of an otherwise invalid sequence.

Khoury *et al.* also consider constraints on invalid sequences, and introduce the notion of "gradation" of solutions [113]. Sequences are arranged on a partial order, independent of the security policy being enforced, which makes it possible to state that some corrective actions are preferable to others. For example, a policy stating that every acquired resource must eventually be relinquished could be enforced by forcibly removing the resource from the control of a principal and reallocating it to another user; a monitor could then seek to allocate the resource equitably between all users, or to minimize the amount of time the resource is idle. In a similar vein, Drábik *et al.* [65] propose to associate each action taken by the monitor with a cost, and to seek optimal cost. Their notion of transparency binds the monitor in its handling of both valid and invalid sequences; it is defined as a function $f : \Sigma^* \to \mathbb{R}$, which the monitor must either maximize or minimize, depending on its formulation. This is the work that is most closely related to the current study.

A few elements stand out in this line of research. First, most approaches impose on the designer to create a finite-state machine that enforces the desired policy and respects any limitations on the capabilities of the monitor (with the exception of [72], which provides a monitor synthesis algorithm). This is a nontrivial task, made even harder when some guarantee of optimal enforcement cost is sought. Furthermore, elaborate proofs are often required to ensure that the enforcement of the property is correct, transparent, and optimal. The use of a fixed cost for each program action is limiting. One may prefer a more flexible gradation of solutions, in which the value associated with a solution is more context-specific.

CHAPTER III
MULTI-TRACES

We group under the term “access restrictions” the conditions that cause a source of events to become imperfectly known by a monitor. In this chapter, we present our formal model to account for access restrictions in a monitoring context. We also review the works surveyed in Section 2.3 focusing on designing monitoring algorithms that are *tolerant* to missing or uncertain events; however, we shall see that they present some limitations in the kind of information degradation they can account for (compared to our approach).

In this chapter, we define an abstract model of access restrictions over event traces. We start by defining an *access proxy*, which is a formal model representing the degradation of events. Our contribution stands out from related works in Section 2.3 in that it intervenes this access proxy between a source of events and a monitor. Each concrete event is modeled as a completely defined “possible world”; the action of the proxy has the effect of potentially turning a unique world into a *set* of such worlds, or deleting events altogether. Obviously, the presence of the proxy and the degradation it causes on the input events have an impact on the verdict produced by the monitor: for instance, it can result in multiple possible verdicts, a phenomenon we call *ambiguity*.

The remaining part of the chapter is structured as follows: Section 3.1 provides a detailed explanation of how we construct our abstract model of access restrictions over event traces. In Section 3.2, we discuss the construction process that extends a classical monitor to a loss-tolerant “multi-monitor” using Mealy machines. Finally, Section 3.3 serves as the concluding section of the chapter.

Note that in Chapter 5, specifically in Section 5.2, we effectively put our conceptual ideas from this chapter into practice by extending a well-known event stream processing engine called BeepBeep [93]. We conduct experiments encompassing various scenarios and subsequently discuss the results.

3.1 TRACE PROXIES

In this section, we describe a formal framework in which the various situations described in Section 2.3 can be modeled. Since our modeling of traces must account for access restrictions in the contents of events in a trace, we must first define an appropriate logical framework for dealing with it. Then, we show how the traditional definition of trace and monitor can be generalized to uncertain or "lossy" traces.

3.1.1 MULTI-TRACES AND PROXIES

Let $\mathbb{B} = \{\bot, \top\}$ be the set of Boolean truth values; let $\mathcal{A} = \{a, b, \dots\}$ be a finite set of atomic propositions. A valuation is a total function $\omega : \mathcal{A} \to \mathbb{B}$ that assigns a truth value to every atomic proposition. For $b \in \mathbb{B}$ and $a \in \mathcal{A}$, we note by $\omega[a \mapsto b]$ the valuation ω' such that $\omega'(x) = \omega(x)$ when $x \neq a$, and $\omega'(a) = b$. We denote by Ω the set of all valuations over $\mathcal{A}$.

In our context, a *uni-trace* is a finite sequence of valuations $\overline{\omega} = \omega_0, \omega_1, \dots, \omega_n$; we denote the set of uni-traces as Ω^*. Valuations, when seen as elements of a trace, will be called "events". The notation $\overline{\omega}[i]$ will be used to denote the event at the i-th position in a trace $\overline{\omega}$. The length of a trace $\overline{\omega}$ will be noted $|\overline{\omega}|$. The concatenation of two finite traces $\overline{\omega}$ and $\overline{\omega}'$ is noted as $\overline{\omega} \cdot \overline{\omega}'$. We shall use a special symbol, ϵ, to designate the empty event, which behaves in the usual way: $\epsilon \cdot \overline{\omega} = \overline{\omega} \cdot \epsilon = \overline{\omega}$. "Uni-trace" corresponds to the concept commonly referred to as a *trace* in the literature on RV. However, our introduction of a more general concept of trace in the following necessitates that the distinction be made explicit.

The modeling of events as sets of Boolean variables may seem primitive at first sight. However, we shall remind the reader that the same argument applies here as for SAT solving, and that such a setting is sufficient to model a very wide range of finite structures, given the proper amount of syntactical sugar. Case in point, one of our implemented scenarios in Section 5.2 models manipulations of a virtual shopping cart containing a set of purchased items, while another models temperature readings by a sensor.

Let Φ' be the set of Boolean propositions that can be built using the classical Boolean connectives over $\mathcal{A}$. The definition of a valuation can be extended to propositional formulas by interpreting Boolean connectives in the usual way. For a given formula φ, a valuation ω is said to be *positive* if $\omega(\varphi) = \top$, and *negative* if $\omega(\varphi) = \bot$. We denote by $[\![\varphi]\!]_\top$ the set of its positive valuations representing the positive interpretation of a propositional formula $\varphi \in \Phi'$. A formula is said to be *uniquely positive* if it has a single positive valuation. The set $\Phi'_\top$ represents the subset of Φ' composed of uniquely positive formulas.

Traces can be generalized to *multi-traces*, where each element is not a single valuation, but rather a set of valuations; the set of multi-traces is noted $(2^\Omega)^*$. Given a multi-trace $\overline{v} \in (2^\Omega)^*$, a uni-projection is a uni-trace $\overline{\omega} \in \Omega^*$ such that $\overline{\omega}[i] \in \overline{v}[i]$ for all i. We shall denote by $\mathcal{U}(\overline{v})$ the set of all uni-projections of a multi-trace. The intuition behind uni-traces and multi-traces is that each event of a uni-trace represents a single, completely defined "world". In contrast, an event of a multi-trace may represent multiple, alternate "possible worlds", where each possible world is an event that carries one value). This vehicle will allow us to represent uncertainty and imprecision about the contents of an event.

As a convention, we shall use the symbol ω to represent a valuation (a uni-event), and v to represent a set of valuations (a multi-event). Given a formula $\varphi \in \Phi'$ and a set of valuations

$v \in 2^\Omega$ we say that v *supports* φ, and note $v \vDash \varphi$, if $v \subseteq [\![\varphi]\!]_\top$. Intuitively, a set of valuations supports a formula φ if it only contains possible worlds where φ holds.

Our next step in the management of uncertainty is to define a special type of transducer on multi-traces.

Definition 1. *Let $\overline{v}, \overline{v}', \overline{v}'' \in (2^\Omega)^*$ be three multi-traces and $v \in 2^\Omega$ be a multi-event. A* multi-trace proxy *is a function $\pi \subseteq (2^\Omega)^* \times (2^\Omega)^*$, such that, if $(\overline{v}, \overline{v}') \in \pi$ and $(\overline{v} \cdot v, \overline{v}'') \in \pi$, then $\overline{v}'$ is a prefix of $\overline{v}''$.*

A multi-trace proxy can be represented as a special type of transducer $\pi : (2^\Omega)^* \longrightarrow (2^\Omega)^*$ that does not rewrite the past: that is, if $\overline{v}$ is a prefix of $\overline{v}'$, then $\pi(\overline{v})$ is a prefix of $\pi(\overline{v}')$. When defined in this manner, it is possible to treat a proxy as a stateful function that can be fed input multi-events one by one, and which produces zero or more output multi-events for each of these inputs. We shall abuse notation and also accept that a proxy reads uni-traces by taking each input uni-event as a singleton multi-event.

To this basic definition, we can further qualify various kinds of proxies depending on additional properties. If $\pi : (2^\Omega)^* \longrightarrow (2^\Omega)^*$ is a multi-trace proxy, and $\overline{v} \in (2^\Omega)^*$ is a multi-trace, we say that π is *deterministic* if there exists a single $\overline{v}' \in (2^\Omega)^*$ such that $(\overline{v}, \overline{v}') \in \pi$. Similarly, π is *k-bounded* if for every $v \in 2^\Omega$ and $\overline{v}, \overline{v}', \overline{v}'' \in (2^\Omega)^*$, if $(\overline{v}, \overline{v}') \in \pi$ and $(\overline{v} \cdot v, \overline{v}'') \in \pi$, $|\overline{v}''| - |\overline{v}'| \leq k$.

When a proxy is deterministic, we shall use the notation $\pi(\overline{v})$ to designate the unique multi-trace $\overline{v}'$ such that $(\overline{v}, \overline{v}') \in \pi$. A proxy is called *world-preserving* when it produces exactly one output event for each input multi-event, and all valuations of the input multi-event are still valuations of the output multi-event, i.e. possible worlds are not removed.

3.1.2 MONITORS FOR MULTI-TRACES

A multi-trace proxy generalizes the notion of a monitor for some abstract property P. In what follows, we override the definition of ϵ to represent an empty trace. Our (propositional) monitor $\pi_P : \Omega^* \rightarrow \{\Omega, \emptyset, \epsilon\}$ is a deterministic transducer on uni-traces; each event of the trace is a valuation that can be seen as the binary encoding of a symbol of an input alphabet. It produces in return the empty trace (ϵ), or the multi-trace made of a single multi-event, Ω or $\emptyset$. These three outcomes represent the usual verdicts produced by a monitor: Ω represents the true verdict, $\emptyset$ the false verdict, and ϵ the inconclusive verdict. One can define a proxy such that for every (uni-)trace $\overline{\omega}$, we have that $(\overline{\omega}, \Omega) \in \pi_P$ if and only if $\overline{\omega}$ satisfies P, $(\overline{\omega}, \emptyset) \in \pi_P$ if and only if $\overline{\omega}$ violates P, and $(\overline{\omega}, \epsilon) \in \pi_P$ otherwise. It is also required that for every $x \in \{\Omega, \emptyset\}$ and every uni-trace $\overline{\omega} \in \Omega^*$, if $\pi_P(\overline{\omega}) = x$, then $\pi_P(\overline{\omega} \cdot \overline{\omega}') = x$ for every $\overline{\omega}' \in \Omega^*$. This corresponds to the intuition that a monitor producing a conclusive verdict for an input trace does not change its verdict when appending events to that trace.

We devise a construction to turn a monitor for uni-traces (a "uni-monitor") into one for multi-traces (a "multi-monitor").

Definition 2. *Let $\pi_P : \Omega^* \rightarrow \{\Omega, \emptyset, \epsilon\}$ be a uni-monitor for some property P. The multi-monitor lifted from π_P, noted $\hat{\pi}_P$, is the multi-trace proxy $\hat{\pi}_M : (2^\Omega)^* \rightarrow 2^{\{\Omega, \emptyset, \epsilon\}}$ such that, for every multi-trace $\overline{v} \in (2^\Omega)^*$: $\hat{\pi}_P(\overline{v}) \triangleq \bigcup_{\overline{\omega} \in \mathcal{U}(\overline{v})} \{\pi_P(\overline{\omega})\}$.*

For a given multi-trace, the output of the multi-monitor is the set of outputs obtained by running the underlying uni-monitor on every possible uni-projection. This set of outputs will be called the *multi-verdict*.

The fact that events fed to a monitor can now contain multiple valuations has an impact on the possible verdicts produced by the monitor. We say that a uni-monitor π_P is *ambiguous* for a multi-trace $\overline{v}$ if $|\hat{\pi}_P(\overline{v})| > 1$. This corresponds to the fact that two uni-projections of $\overline{v}$ result in two different verdicts. The monitor is *strongly ambiguous* for $\overline{v}$ if $\{\Omega, \emptyset\} \subseteq \hat{\pi}_P(\overline{v})$; in such a case, the monitor produces two contradictory verdicts. Otherwise, the monitor is called *weakly* ambiguous (for example, if $\hat{\pi}_P(\overline{v}) = \{\Omega, \epsilon\}$). In the case where a monitor is ambiguous for a given multi-trace, we can provide a measure of this ambiguity; each verdict can be associated to the fraction of all uni-traces that yield this verdict, and hence be used as a quantitative indication of its likelihood.

For $v_r \in \{\Omega, \emptyset, \epsilon\}$, we define a function $\rho^{v_r}_{\pi_P} : (2^\Omega)^* \to [0, 1]$ as:

$$\rho^{v_r}_{\pi_P}(\overline{v}) \triangleq \frac{|\{\overline{\omega} \in \mathcal{U}(\overline{v}) : \pi_P(\overline{\omega}) = v_r\}|}{|\mathcal{U}(\overline{v})|}$$

The function $\rho^{v}_{\pi_P}$ represents the fraction of all uni-projections of $\overline{v}$ for which the monitor π_P produces the verdict v_r.

We shall now consider a binary system composed of an access proxy and a monitor, in such a way that the output of the first is given as the input to the second.

Definition 3. *Let $\pi_A : \Omega^* \to (2^\Omega)^*$ be a proxy that turns uni-traces into multi-traces, and $\pi_P : \Omega^* \to \{\Omega, \emptyset, \epsilon\}$ is a uni-monitor as defined earlier. The access-controlled monitor $\mathcal{M}(\pi_A, \pi_P) : \Omega^* \to 2^{\{\Omega, \emptyset, \epsilon\}}$ is the trace proxy defined as $\mathcal{M}(\pi_A, \pi_P) \triangleq \hat{\pi}_P \circ \pi_A$.*

The intuition between an access-controlled monitor is that uni-events are produced from some abstract source; these uni-events are then transformed by the action of the proxy π_A, resulting in a multi-trace. Hence, π_A represents the "degradation" of the original uni-trace. This multi-trace is then fed to the multi-monitor lifted from π_P, and its set of verdicts represents the output of the access-controlled monitor.

We can then extend the definition of ambiguity to an access-controlled monitor. Given a uni-monitor π_P for some property P and an access control proxy π_A, we say that π_P is *strongly* (resp. weakly) *affected* by π_A if there exists a uni-trace for which $\mathcal{M}(\pi_A, \pi_P)$ is strongly (resp. weakly) ambiguous. Finally, a monitor π_P is called *sound under* π_A if, for every uni-trace $\overline{\omega}$, the verdict of $\mathcal{M}(\pi_A, \pi_P)$ contains the verdict of π_P. It is easy to see that world preservation is a sufficient condition for soundness:

Theorem 1. *If π_A is world-preserving, then $\hat{\pi}_P$ is sound under π_A.*

Proof. Let $\overline{\omega}$ be a uni-trace. Since π_A is world-preserving, one of the uni-projections of $\pi_A(\overline{\omega})$ is $\overline{\omega}$ itself. Combining definitions 2 and 3, it follows that $\pi_P(\overline{\omega}) \in \hat{\pi}_P(\pi_A(\overline{\omega}))$. ■

3.1.3 MODELING ACCESS RESTRICTIONS WITH PROXIES

Equipped with these basic definitions, we can now illustrate how the concept of access proxy can be used to model the various use cases about imprecise and uncertain data enumerated in Section 2.3, including situations that cannot be accounted for in existing models of "lossy RV" discussed earlier.

3.1.3.1 MISSING, CORRUPTED, AND ENCRYPTED VALUES AND EVENTS

Missing values can first be modeled by altering the set of valuations of an input event. Consider for example the proxy π_1 defined as $\pi_1(v_1, \dots, v_n) \triangleq \pi_1(v_1, \dots, v_{n-1}) \cdot f_1(v_n)$, where $f_1 : 2^\Omega \to 2^\Omega$ is defined as:

$$f_1(v) \triangleq \bigcup_{\omega \in v} \{\omega[a \mapsto \top], \omega[a \mapsto \bot]\}$$

The action of function f_1 can be explained as follows: whatever the input multi-event $v \in 2^\Omega$, in the output event $f_1(v)$ we have neither $f_1(v) \models a$ nor $f_1(v) \models \neg a$ (all other variables being left unchanged).[1] In other words, in the output event, we can no longer conclude anything about the value of a in the input event. This is equivalent to a representation of uncertainty using a third "unknown" Boolean value [27]. It can be used to represent the fact that one of the readings inside an event is corrupted, missing, or encrypted with a key that is not in the possession of the recipient. In the case where each event represents a set of observations at a given time point, this proxy can also represent the fact that it is unknown whether a occurred or not in a time point.

An extreme case is a known missing event —that is, an event whose occurrence is known or has been deduced (for example by observing gaps in indexes, or after a database request has been denied), but whose content is completely missing. This can be represented in our model by an event that contains all valuations, i.e. Ω. The case of load shedding discussed in Section 2.2.2 can be modeled using such a mechanism, which is equivalent to the non-event χ used by [105] precisely to account for this situation.

For example, consider the proxy π'_1 defined as

$$\begin{aligned}
\pi'_1(v) &\triangleq v \\
\pi'_1(v_1, \dots, v_n) &\triangleq \pi'_1(v_1, \dots, v_{n-1}) \cdot v_n \text{ if } v_{n-1} \neq v_n \\
\pi'_1(v_1, \dots, v_n) &\triangleq \pi'_1(v_1, \dots, v_{n-1}) \cdot \Omega \text{ otherwise}
\end{aligned}$$

This proxy reduces the length an input trace by replacing any stuttering events by Ω, symbolizing a deleted event. Alternately, a proxy could emit and discard events in an alternating fashion, to

[1]The condition is actually even stronger: for any formula φ such that $v \models \varphi$, we have neither $\varphi \rightarrow a$ nor $\varphi \rightarrow \neg a$.

represent a form of systematic preemptive load shedding. One can also imagine variations over this basic mechanism: for example, the proxy could output all events until the occurrence of some trigger that activates load shedding, while some other trigger returns the proxy to normal operation.

3.1.3.2 UNCERTAINTY AND FUZZINESS

Values inside an event may not be completely unknown, and only involve some amount of fuzziness. This is especially the case for sensor readings, where numerical values are typically accompanied by a precision interval. A discrete set of numerical values $\mathbb{D}$ can be modeled with Boolean variables in various ways —an easy one being to associate each value $d \in \mathbb{D}$ to a Boolean variable b_d. Uncertainty can then be represented as a function $\gamma : \mathbb{D} \to 2^{\mathbb{D}}$.

A proxy π_2 can be defined as $\pi_2(v_1, \ldots, v_n) \triangleq \pi_2(v_1, \ldots, v_{n-1}) \cdot f_2(v_n)$, where $f_2 : 2^{\Omega} \to 2^{\Omega}$ is defined as:

$$f_2(v) \triangleq \bigcup_{\omega \in v} \bigcup_{b \in \gamma(b_v)} \{\omega[b_v \mapsto \bot, b \mapsto \top]\}$$

where b_v is the unique b_i in ω such that $b_i = \top$. In other words, the proxy turns each valuation where b_d is true into the set of valuations where each $b_i \in \gamma(d)$ is made successively true (and leaves any other variables unchanged). This corresponds to the intuition that in any possible world, the numerical value can be any one of $\gamma(d)$, but only one of them at a time and none of the other values. Stated in this way, it is the *discrete* equivalent of the notion of *abstract data domain* in Abstract TeSSLa's modeling of uncertainty [125].

Note however that this form of imprecision cannot be accounted for in a model where each event is a single possible world with ternary Boolean values (i.e. [27]). Giving the value “unknown” to all temperature variables in $\gamma(d)$ misses the fact that in any possible interpretation, exactly one of them *must* be the value. In other words, this modeling is an over-approximation that introduces the spurious possible world where the event can contain “no value”, or many values. This situation cannot be modeled either by Joshi *et al.*’s approach [105], where events are atomic and are either completely known or completely unknown (except for their occurrence).

3.1.3.3 CORRELATED UNCERTAINTY AND FUZZINESS

So far, the examples of degradation we have shown apply in an independent manner to a single input variable or signal at a time. Correlated uncertainty occurs when deterioration of information is applied in a way that depends on more than one input variable.

Consider the proxy π_3 defined in the same way as π_2, but with f_2 replaced by $f_3(v) \triangleq v \cup \bigcup_{\omega \in v} \{\omega_{a \leftrightarrow b}\}$. The notation $\omega_{a \leftrightarrow b}$ designates the valuation that swaps the assignments of a and b in ω. This has the effect of making a and b indistinguishable: an input multi-event that supports a is transformed into an output multi-event that only supports the weaker proposition $a \vee b$ (and similarly for events that support b). In other words, it is no longer possible to conclude precisely that a is true or that b is true, only that at least one of them is true. This is a simple form of the impedance mismatch use case we discussed in Section 2.2.3.

This situation cannot be accounted for in any of the models we surveyed in Section 2.3. In three-valued logic, the reasoning is the same as before: the best one can do in such a model is to over-approximate uncertainty by stating that the occurrence of both a and b is unknown (this abstraction is still precise for events where neither a nor b are true). For abstract data domains [125], the situation becomes even less desirable: since these domains are defined for each variable separately, and must remain the same for the entire trace, the only world-preserving abstraction is the one that replaces values of a and b with both their possible values at all time points, which is an even greater over-approximation.

3.2 AUTOMATA-BASED TRACE PROXIES

As one can see, the multi-event model and the definition of a trace proxy are very flexible in their ability to model various forms of imprecision, uncertainty, and missing or incorrect values. In the following, we will focus on one specific representation of multi-trace proxies by providing an extension of Mealy machines.

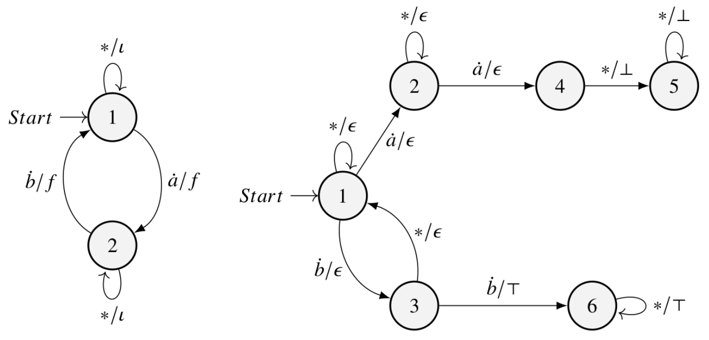
3.2.1 PROPOSITIONAL MEALY MACHINES

In the following, we shall assume that the representation of trace proxies is based on a special type of finite-state machine called a *propositional mealy machine*. It is formally defined as follows.

Definition 4. *A* propositional machine *is a triplet $M = \langle S, s^0, \mu\rangle$ consisting of a finite set of states S, a unique start state $s^0 \in S$, and a marking $\mu \subseteq S \times \Phi \times S$ associating propositional formulas to pairs of states.*

Figure 3.1 shows two examples of propositional machines represented graphically. As one can see, the main difference between a propositional machine and a traditional finite-state machine is the fact that input symbols and transitions are replaced by a marking with *propositional formulas*. Note that there can be two formulas φ, φ' associated with the same two states; these would be represented as two distinct edges in the graph representation of the machine.

The figure illustrates a few notational shortcuts we shall use in the following. Given



(a) Proxy π_A (b) Monitor π_P

Figure 3.1 : Access-controlled monitor represented as a pair of propositional machines.

propositional variables $\mathcal{A} = \{a_1, \ldots, a_m\}$, the notation $\dot{a}_k$ represents the propositional formula $(\bigwedge_{i \neq k} \neg a_i) \wedge a_k$. Moreover, the special symbol $*$ is meant as a notation shortcut meaning "otherwise": in a given state, it corresponds to the propositional formula made by the conjunction of the negation of all formulas in the other outgoing edges. For example, assuming that $\mathcal{A} = \{a, b, c\}$, the $*$ symbol in state 1 of Figure 3.1a corresponds to the formula $\neg a \vee b \vee c$ (which is the negation of $\dot{a} = a \wedge \neg b \wedge \neg c$).

Definition 5. *Let $\mu \subseteq S \times \Phi \times S$ be a marking over a propositional machine M. The transition relation induced by μ, is the relation $\tilde{\mu} \subseteq S \times 2^{\Omega} \times S$ such that, for every $(s, \varphi, s') \in \mu$ and every $v \in 2^{\Omega}$, we have that $(s, v, s') \in \tilde{\mu}$ if and only if $v \cap [\![\varphi]\!]_{\top} \neq \emptyset$.*

Intuitively, in a state s and given an input multi-event $v \in 2^{\Omega}$, the transition $s - \varphi \longrightarrow s'$ is possible through v if the positive valuations of φ contain at least one valuation of v. In other words, there exists one "possible world" admitted by φ that is compatible with v. In turn, this definition of a transition relation can be lifted to multi-traces expressed as sequences of Boolean formulas; given a multi-event $\varphi' \in \Phi'$ as an input Boolean formula, the transition $s - \varphi \longrightarrow s'$ is possible if $[\![\varphi]\!]_{\top} \cap [\![\varphi']\!]_{\top} \neq \emptyset$. This corresponds to the situation where both φ and φ' share at least one positive valuation.

Boolean formulas are a useful shortcut to succinctly represent sets of valuations. In the following, we shall concentrate on multi-traces viewed as elements of Φ'^*. Given a multi-trace $\overline{\varphi}$ of length n, a possible *run* of $\overline{\varphi}$ in M is any sequence $s_0 - \psi_0 \longrightarrow s_1 - \psi_1 \longrightarrow \cdots - \psi_{n-1} \longrightarrow s_n$ such that $[\![\varphi[i]]\!]_\top \cap [\![\psi_i]\!] \neq \emptyset$ and $(s_i, \psi_i, s_{i+1}) \in \mu$ for every $i \in [0, n-1]$. In such a case, the complexity of determining if a sequence of transitions is a run for some $\overline{\varphi} \in \Phi'^*$ can be precisely established.

Theorem 2. *Let $\mathcal{A} = \{a_1, \ldots, a_m\}$ be a set of m propositional variables. Let $\overline{\varphi} = \varphi_0, \ldots, \varphi_{n-1}$ be a finite multi-trace over $\mathcal{A}$ and M be a propositional machine. Let $s_0, \ldots, s_n$ be a sequence of states, and ψ_0, ..., ψ_{n-1} a sequence of transition labels in μ. Determining if $s_0 - \psi_0 \longrightarrow s_1 - \psi_1 \longrightarrow \cdots - \psi_{n-1} \longrightarrow s_n$ is a run of M for $\overline{\varphi}$ is NP-complete.*

Proof. Let ψ be a propositional formula. We create the propositional machine made of a single state s, with a single transition $s - \psi \longrightarrow s$. Let $\overline{\varphi} = \top$ be the multi-trace made of the single multi-event $\top$. By Definition 5, the sequence $s - \psi \longrightarrow s$ is a run of M for the trace made of the single multi-event ψ if and only if $[\![\top]\!]_\top \cap [\![\psi]\!]_\top \neq \emptyset$, i.e. if ψ is satisfiable. This shows that the problem is NP-hard.

Let $\psi_0, \ldots, \psi_{n-1}$ be the sequence of formulas such that ψ_i is the formula associated to the transition $s_i \longrightarrow s_{i+1}$ in M. For each i, define the set of propositional variables A_i as $\{a_1^i, \ldots, a_m^i\}$. Given a formula φ over $\mathcal{A}$, the renaming of $\mathcal{A}$ to $\mathcal{A}^i$, noted $\rho_i(\varphi)$, is the propositional formula obtained by replacing a_j by a_j^i for every $j \in [1, m]$. Let $\hat{\varphi}$ be the propositional formula defined as:

$$\hat{\varphi} \triangleq \bigwedge_{i=0}^{n-1} (\rho_i(\varphi_i) \wedge \rho_i(\psi_i))$$

The sequence $s_0 - \psi_0 \longrightarrow \cdots - \psi_{n-1} \longrightarrow s_n$ is a run of M for $\overline{\varphi}$ if and only if $\hat{\varphi}$ is satisfiable, hence the problem is in NP. ■

In the general case, a propositional machine can have multiple runs, not necessarily one unique run, even when the formulas on each outgoing transition in a state are mutually exclusive. Case in point, consider the machine of Figure 3.1b; in state 1, if the machine receives for its input event the proposition $a \vee b$, one can see that the input event can fire both the transitions $\dot{a}$ and $\dot{b}$, and therefore both 2 and 3 are possible next states.

Definition 6. *Let $s - \psi \rightarrow s'$ be a transition in M and let $\ell : \Phi' \rightarrow \Phi'$ be a function that transforms an input multi-event into another output multi-event. The function $\gamma : S \times \Phi' \times S \rightarrow (\Phi' \rightarrow \Phi)$ associates the function $\ell : \Phi' \rightarrow \Phi'$ to the transition $s - \psi \rightarrow'$ to produce an output symbol.*

Given an input multi-trace that induces a run in M, the resulting output multi-trace is defined as follows:

Definition 7. *Let $\overline{\varphi} = \varphi_0, \dots, \varphi_{n-1}$ be a finite multi-trace and let $s_0 - \psi_0 \rightarrow s_1 - \psi_1 \rightarrow \cdots - \psi_{n-1} \rightarrow s_n$ be a run of M for $\overline{\varphi}$. The output multi-trace produced by M is the sequence $\overline{\varphi}' = \varphi'_0, \dots, \varphi'_{n-1}$, where $\varphi'_i = \gamma(s_i, \psi_i, s_{i+1})(\varphi_i)$ for each $i \in \{0, \dots, n-1\}$.*

Simply put, an input formula is replaced by applying to it the function $\ell : \Phi' \rightarrow \Phi'$ associated by γ to the corresponding transition in M. This function effectively turns a propositional machine into an extended version of a Mealy machine. We can further extend the definition of ℓ, and allow its image to be $\Phi' \cup \{\epsilon\}$. In such a case, the machine may produce no output (ϵ) for an input event. This representation makes it possible to model the class of *1-bounded* trace proxies.

In the following, we will represent a few functions $\ell : \Phi' \rightarrow \Phi'$ by special symbols. Function ι will designate the identity, i.e. $\iota(\varphi) = \varphi$ for all $\varphi \in \Phi'$. For a given formula φ, we will abuse notation and use φ to designate the constant function that turns any input formula into φ (the most notable ones being the constants $\top$ and $\bot$). Although ℓ accepts and returns a Boolean formula, it is not excluded that its definition be made in terms of sets of valuations. For example, the proxy π_3, defined in Section 3.1.1 could be defined by first converting an input formula φ into its set of positive valuations $[\![\varphi]\!]_\top$, performing the transformations on that set, and converting this set back into a Boolean formula φ'.[2]

To illustrate this point, Figure 3.1 shows a simple example of a pair of access proxy and monitor, on the input alphabet $\mathcal{A} = \{a, b, c\}$. In this case, the function f in π_A is the function f_3 already given as an example in Section 3.1.2, which makes it impossible to know which one of a or b is true in an input event, only that at least one of them is true. This has an impact on the verdict that can be produced by π_P for some of the traces it receives. For example, the uni-trace $\dot{a}, \dot{b}, \dot{c}, \dot{a}$ is transformed by the access proxy into $\dot{a} \vee \dot{b}, \dot{a} \vee \dot{b}, \dot{c}, \dot{a} \vee \dot{b}$. There are 8 possible runs in π_P for this input multi-trace, including one that visits the states 1–2–4–5, and produces the verdict $\bot$, and another that visits the states 1–3–6–6, and produces the verdict $\top$. Therefore, the verdict becomes ambiguous.

3.2.2 A MONITORING ALGORITHM

Equipped with such definitions, we can now define an algorithm which, given an access-controlled monitor $\mathcal{M}(\pi_A, \pi_P)$ expressed as a pair of propositional machines and a finite multi-trace $\overline{\varphi} \in \Phi'^*$, computes the multi-verdict associated to this prefix. Furthermore, this multi-verdict is quantified —that is, if its output set contains more than one value, the fraction computed by ρ, as defined in Section 3.1.3.3, will be associated to each value.

The procedure is defined in Algorithm 1, for an access proxy $\pi_A = \langle s_A^0, S_A, \mu_A \rangle$ and a

[2]One easy way being by creating the disjunction of each valuation.

Algorithm 1 Access-controlled state update algorithm

```
procedure UPDATE($\varphi, s_A, \sigma$)
    $\sigma' \leftarrow \emptyset$                                            ▷ $\sigma' : S_P \rightharpoonup \mathbb{N}$
    $\beta \leftarrow \emptyset$                                             ▷ $\beta : \{\Omega, \emptyset, \epsilon\} \rightharpoonup \mathbb{N}$
    $(\psi_A, s'_A) \leftarrow$ the unique $\psi_A, s'_A$ s.t. $(s_A, \psi_A, s'_A) \in \tilde{\mu}_{\pi_A}$
    $\ell_A \leftarrow \gamma_{\pi_A}(s_A, \psi_A, s'_A)$
    $\varphi' \leftarrow \ell_A(\varphi)$
    for $(s_P, n) \in \sigma$ do
        for $(s_P, \psi_P, s'_P) \in \tilde{\mu}_{\pi_P}$ do
            $c \leftarrow |[\![\psi_P]\!]_\top \cap [\![\varphi']\!]_\top|$
            if $c > 0$ then
                $\sigma'(s'_P) \leftarrow \sigma'(s'_P) + (n * c)$
                $l \leftarrow \gamma_{\pi_P}(s_P, \psi_P, s'_P)$
                $\beta(l) \leftarrow \beta(l) + (n * c)$
            end if
        end for
    end for
    return $\langle s'_A, \sigma', \beta \rangle$
end procedure
```

uni-monitor $\pi_P = \langle s_P^0, S_P, \mu_P \rangle$. We assume that π_A and π_P are both deterministic and that their transition relation is total. The algorithm takes as input a multi-event $\varphi \in \Phi'$, a state $s_A \in S_A$, and a partial function $\sigma : S_P \rightharpoonup \mathbb{N}$. Intuitively, φ is the new input event to ingest, s_A is the current state in π_A reached after reading a trace prefix $\overline{\varphi}$ (note that the Update function is called each time an event φ is processed. It reads the whole trace $\overline{\varphi}$ event by event, and the state s_A represents the current state after reading all the preceded events before the current event φ being processed), and for some $s \in S_P$, $\sigma(s)$ designates the number of uni-projections of $\overline{\varphi}$ that result in a run of π_P ending in state s ($\sigma(s)$ being undefined can be assimilated to the case $\sigma(s) = 0$, which indicates that no uni-projection ends in s).

Lines 2–3 initialize an empty partial function $\sigma' : S_P \rightharpoonup \mathbb{N}$, and an empty partial function $\beta : \{\Omega, \emptyset, \epsilon\} \rightharpoonup \mathbb{N}$. Function σ' stores the update of σ after ingesting the input event; β maps each of the three verdicts to the number of uni-projections being mapped by π_P to that verdict. Line 4 identifies the transition $(s_A, \psi_A, s'_A) \in \tilde{\mu}_{\pi_A}$ that can be taken from s_A and input event φ; since we assumed that φ is a uni-event and that π_A is total and deterministic, this transition exists and is unique. Lines 5–6 then apply the output function $\gamma_{\pi_A}(s_A, \psi_A, s'_A)$ that associates the transformation function ℓ_A to the transition producing the resulting output multi-event φ'. Considering the uni-trace $\dot{a}, \dot{b}, \dot{c}, \dot{a}$ of Figure 3.1, processing the event $\dot{b}$ takes the transition (2,$\dot{b}$,1) and emits the output $\dot{a} \vee \dot{b}$.

The second part of the algorithm is made of the lines 7–16, and corresponds to the update of both the uni-monitor's states, and the count of uni-projections for each verdict. The algorithm takes in succession each state $s_P \in S_P$ of the uni-monitor reached after processing one of the uni-projections of $\overline{\varphi}$. From each such state s_P, it computes the count c of uni-projections that can fire the condition ψ_P on each outgoing transition. When this is the case, the number of uni-projections reaching state s'_P, stored in σ', is incremented by nc (line 11), where n is the number of uni-projections of $\overline{\varphi}$ reaching s_P. This calculation can be explained by the fact that, if there are n uni-projections of $\overline{\varphi}$ that reach s_P, and c uni-projections of φ allow us to take the transition $s_P - \psi \longrightarrow s'_P$, then there are nc uni-projections of $\overline{\varphi} \cdot \varphi$ whose last two visited states are s_P and s'_P. For example, the multi-monitor π_P of Figure 3.1 processes each output emitted by the proxy π_A and selects all the possible output transitions (uni-projections) that can be taken in π_P. Starting by the multi-event $\dot{a} \vee \dot{b}$, it takes two possible output transitions ($1 \longrightarrow 2$ and $1 \longrightarrow 3$). Reaching the event $\dot{b}$, it processes the multi-event $\dot{a} \vee \dot{b}$ and selects two possible output transitions: transition $2 \longrightarrow 4$ and transition $3 \longrightarrow 6$. Then, the total number of uni-projections reached after processing $\dot{a} \vee \dot{b}$ is two: 1–2–4 and 1–3–6.

The verdict l produced by π_P on taking the transition $s_P - \psi \longrightarrow s'_P$ is fetched (line 12). Then, mapping β is updated: the number $\beta(l)$ of uni-projections producing verdict l is incremented by nc (line 13), by the same reasoning as for the update of σ'. The process repeats for all states $s_P \in S_P$ defined in σ. Upon termination, the algorithm returns the triplet $\langle s'_A, \sigma', \beta \rangle$: s'_A is the new current state of π_A, σ' maps each possible current state of π_P with a number of uni-projections, and β does the same with each verdict.

In order to compute the verdict of an access-controlled monitor $\mathcal{M}(\pi_A, \pi_P)$ on a uni-trace $\overline{\varphi}$, it suffices to call the procedure UPDATE repeatedly. A straightforward procedure called MONITOR (not shown due to lack of space) can iterate over each event in $\overline{\sigma}$, call UPDATE repeatedly, and output the current mapping β associating each of the three verdicts to the corresponding number of uni-projections. The start configuration of this procedure is simply the unique initial state s_A^0 and the mapping that stipulates that a single uni-projection of the empty trace reaches the unique initial state of π_P. The following theorem states that the output map produced by MONITOR on a uni-trace $\overline{\varphi}$ does correspond to the number of uni-projections of $\pi_A(\overline{\varphi})$ that result in each of the three possible verdicts.

Theorem 3. *Let $\mathcal{M}(\pi_A, \pi_P)$ be an access-controlled monitor expressed as two propositional machines, $\overline{\varphi}$ be a uni-trace and $\beta : \{\Omega, \emptyset, \epsilon\} \to \mathbb{N}$ be the mapping produced by calling MONITOR($\overline{\varphi}$). For $v \in \{\Omega, \emptyset, \epsilon\}$, we have that:*

$$\frac{\beta(v)}{\sum_{v' \in \{\Omega, \emptyset, \epsilon\}} \beta(v')} = \rho^v_{\pi_P}(\pi_A(\overline{\varphi}))$$

Proof. Let φ be a multi-event, $s_A \in S_A$, $s_P \in S_P$ be states in the access proxy and the multi-monitor, and $\sigma : S_P \to \mathbb{N}$. For some multi-trace $\overline{\varphi} = \varphi_0, \ldots, \varphi_{n-1}$ and every state $s \in S_P$, $\sigma(s)$ is the number of runs of the form $s_0 - \varphi_0 \to \ldots - \varphi_{n-1} \to s$ in π_P. For every $s' \in S_P$, $\sigma'(s')$ is the number of runs of the form $s_0 - \varphi_0 \to \ldots - \varphi_{n-1} \to s' - \varphi \to s''$ in π_P.

Moreover, we have that for every iteration of lines 8–15, $nc = \sigma(s_P) + |[\![\psi_P]\!]_\top \cap [\![\varphi']\!]_\top|$; i.e. nc is the number of runs of the form $s_0 - \varphi_0 \to \ldots - \varphi_{n-1} \to s_P - \varphi \to s'_P$. We can finally observe that for a given verdict $l \in \{\Omega, \emptyset, \epsilon\}$, $\beta(l)$ is the sum of all values nc in iterations where $\gamma_{\pi_P}(s_P, \psi_P, s'_P) = l$. In other words, $\beta(l)$ is the number of runs of $\overline{\varphi} \cdot \varphi$ in π_P that end up in a state labeled with verdict l. Then $\frac{\beta(v)}{\sum_{v' \in \{\Omega, \emptyset, \epsilon\}} \beta(v')}$ is the fraction of all runs that end up in this verdict, which is equal to $\rho^{v}_{\pi_P}(\pi_A(\overline{\varphi}))$. ■

3.2.3 DISCUSSION

A few remarks must be made about this algorithm. First, it operates “on-the-fly”: each new input event is handled by updating states and uni-projection counts obtained on the previous computation step. In other words, the algorithm does not need to recalculate everything from the start, which makes it possible to operate in streaming fashion. In particular, it does not explicitly enumerate all uni-projections. Second, it merges the operation of the access proxy π_A and the monitor π_P. An algorithm only for the multi-monitor lifted from π_P (i.e. $\hat{\pi}_P$) can easily be obtained by removing lines 2–6 of Algorithm 1 and using φ' as the input to Update.

In terms of complexity, a call to Update is dominated by the loop in lines 7–16; one can easily see that the number of iterations of the inner loop of lines 9–14 is bounded by $|S_P| \cdot |\tilde{\mu}_{\pi_P}|$. However, each such iteration involves an execution of line 9, which computes the number of positive valuations that are common to two Boolean formulas. A naïve way of obtaining this count it is to enumerate all $2^{|\mathcal{A}|}$ valuations. From this, we can conclude that the complexity of Monitor is linear in the length of the input trace, the number of states and the number of transitions of π_P, and exponential in the number of propositional variables encoding each event.

The fact that each call to the monitor involves solving multiple NP-complete problems may seem alarming. However, this is mitigated by two observations. First, SAT instances typically involve very large numbers of variables. It is expected that the Boolean encoding of events, in a monitoring context, will be much smaller. For example, a monitoring problem with an alphabet of 1,000 different events can be encoded using only 10 Boolean variables. Therefore, the model counting and SAT problems involved are expected to be, in comparison, very small.

3.3 CONCLUSION OF THE CHAPTER

In this chapter, we introduced a versatile framework for addressing limitations on event access within a trace. We employed a stateful proxy to represent the existing gaps and imprecise values in the events, while also incorporating various forms of uncertainty. This modified event stream was then passed to the monitor. Additionally, we proposed a method to construct a multi-monitor capable of handling losses from a single monitor while maintaining functionality.

A first advantage of our model is that it makes possible, for a given input trace and a monitor, to study the effect of various kinds of degradation on the monitor's verdict. It is also flexible: the manipulations made to the input trace can be stateful (i.e. the alteration applied to an event, if any, may depend on the past), and the "multi-events" resulting from an input event can account for various types of data degradation and access limitations, including some that cannot be modeled by existing related works.

Yet another advantage of our abstract model is that, contrary to existing works, it is agnostic to the concrete way in which the proxy and the monitor are specified. In Section 3.2, we present one such possible way, by defining an extension of Mealy machines where symbols for transitions and outputs are replaced by logical formulas. We describe a construction that lifts a loss-tolerant "multi-monitor" from a classical monitor. In Chapter 5, we shall see that our loss-tolerant monitor runs in linear time in the size of the trace and the size of the underlying monitor. In the case where an imprecise trace leads to more than one possible verdict, it quantifies the likelihood of each possible verdict on-the-fly.

CHAPTER IV
ENFORCEMENT

In this chapter, we introduce a model of runtime enforcement composed of three separate stages. The first stage transforms events of an (invalid) input trace into a set of traces, obtained by applying each possible modification one is allowed to apply. The second stage filters this set to keep only the traces that do not violate a specified security policy, while the third stage ranks the remaining traces based on an objective gradation, which we term the *enforcement preorder*, and picks the highest-scoring trace as its output.

This design provides a high level of modularity. First, the expression of the allowed modifications to the trace, the security policy itself and the enforcement preorders can all be expressed independently, using a different formal notation if needed. This, in turn, makes it easier to reason about the behavior of the whole pipeline. Second, the model does not require a specific EM to be manually synthesized for each policy to enforce: corrective actions are computed, selected and applied dynamically. Finally, the model does not impose a single valid output; rather, it allows multiple corrective actions to be compared against the enforcement preorder provided by the user.

This chapter is organized as follows: Section 4.1 introduces the notion of proxy as a transducer and describes categories of proxies used in the literature. Section 4.2 describes the notion of trace correction as an interplay between a monitor output and the alterations made by proxy to correct a trace. Equipped with these notions, we present our model of runtime enforcement in Section 4.3 while illustrating the flexibility of the approach with two use cases adapted from the literature. Concluding remarks are given in Section 4.4.

In section 5.3 of Chapter 5, we present a concrete implementation of our pipeline and different categories of proxies as extensions of the BeepBeep event stream processing library [93] and provide a comparison and discussion of the obtained results.

4.1 ALTERING INPUT TRACES

In the situation where an input trace of events violates the policy, one may consider the possibility of modifying this trace so that it becomes compliant with a security property. In order to do so, we must first formalize how these modifications are applied. In this section, we introduce the notion of *transducers* followed by the notion of *proxy* which is a transducer allowing to turn an input trace of events into one or more "modified" versions. We then enumerate particular categories of proxies corresponding to enforcement mechanisms from past literature, and describe a few possible notations to define such proxies.

4.1.1 TRANSDUCERS

Given two sets of events, Σ_1 and Σ_2, a trace transducer is a function $\tau : \Sigma_1^* \to \Sigma_2^*$, with the added condition that for every $\overline{\sigma}, \overline{\sigma}' \in \Sigma^*$, $\overline{\sigma}' \preceq \tau(\overline{\sigma})$ implies $\overline{\sigma}' \preceq \tau(\overline{\sigma} \cdot x)$ for every $x \in \Sigma_1$. In other words, a transducer takes as input a sequence of events, and progressively outputs another sequence of events. Given an arbitrary transducer $\tau : \Sigma_1^* \to \Sigma_2^*$ and a sequence $\overline{\sigma} \in \Sigma_1^*$, we define $\tau_{\overline{\sigma}} : \Sigma_1^* \to \Sigma_2^*$ as $\tau_{\overline{\sigma}}(\overline{\sigma}'') = \tau(\overline{\sigma} \cdot \overline{\sigma}'')$. Intuitively, $\tau_{\overline{\sigma}}$ is a device abstracting the "internal state" of the transducer τ after ingesting the events from the prefix $\overline{\sigma}$.

A transducer is said to be k-bounded if for every $\overline{\sigma} \in \Sigma^*$ and every $\sigma \in \Sigma^*$, $|\tau(\overline{\sigma} \cdot \sigma)| - |\tau(\overline{\sigma})| \leq k$. This means that for every new input event, the transducer adds to its output at most k events. The transducer is said to be k-monotonic if it produces exactly k output events for each input event.

One can lift a policy φ and its associated verdict function $\varphi(\overline{\sigma})$ into a transducer

$\widehat{\phi} : \Sigma^* \to \mathbb{B}_4^*$, defined as follows, for all $i \in \mathbb{N}$:

$$\widehat{\phi}(\overline{\sigma})[i] \triangleq \varphi(\overline{\sigma}[..i])$$

In other words, the i-th output event of $\widehat{\phi}$ corresponds to the verdict associated with $\overline{\sigma}$ after reading its first i events. This transducer is called the *monitor*: it can be seen as an entity observing the input sequence of events and producing after each event the verdict associated with that sequence according to the underlying policy. Under the definition of φ, a monitor's output consists of a sequence of uncertain verdicts (positive, negative, or a mixture of both), and may eventually settle on a definitive true or false verdict, after which it never changes.

4.1.2 PROXIES

Formally, a *proxy* is a 1-monotonic transducer $\pi : \Sigma^* \to (2^{(\Sigma^*)})^*$. It takes as input a trace of events, and outputs a sequence of sets of traces $S_1, S_2, \dots$, with the additional hypothesis that each of the S_i are finite sets of finite traces. We add the soundness condition that: 1) for every $i > 0$ and every $\overline{\sigma} \in S_i$, there exists a $\overline{\sigma}' \in S_{i-1}$ such that $\overline{\sigma}' \preceq \overline{\sigma}$, and 2) for every $\overline{\sigma}' \in S_{i-1}$, there exists $\overline{\sigma} \in S_i$ such that $\overline{\sigma}' \preceq \overline{\sigma}$.

Intuitively, the proxy can be seen as a device that ingests an input sequence of events and outputs after each input event a set of sequences S_i; this set corresponds to the sequences that the proxy "suggests" in replacement of the first i events of the original input trace. In this context, the soundness condition is easier to understand: it corresponds to the fact that a proxy is allowed to extend, possibly in more than one way, any trace that was present in its previous set (including the empty trace), however; it is not allowed to take back a trace that was proposed previously. Since a trace produced by a proxy is actually made of multiple traces, we call it a *multi-trace*.

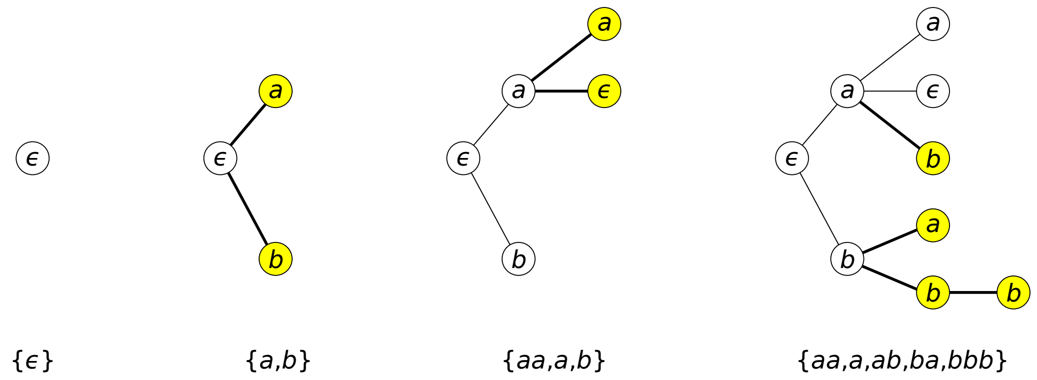


Figure 4.1 : A sequence of prefix trees depicting the output of an arbitrary proxy, where each addition with respect to the previous tree is highlighted. At the bottom are written the explicit traces corresponding to each prefix tree.

This model can be illustrated graphically in the form of a *prefix tree* such as the ones depicted in Figure 4.1, where nodes of the tree are labelled with events. Each path from the root of the tree to one of its leaves represents one possible trace. The proxy starts from a tree made of a single root node; upon receiving each input event, it has the freedom to append any number of nodes to any node of the current tree. This is represented by the sequence in the figure, where additions according to the previous step are highlighted.

The sequence corresponds to the sets of traces $\{a, b\}$, $\{aa, a, b\}$ and $\{aa, a, ab, ba, bbb\}$, respectively. A few observations must be made on this sequence. First, note that it follows the soundness condition expressed above, in that every sequence in a set is the prefix of some sequence in the set that comes after. Also note that the ϵ node, representing the empty sequence of event, needs to be added to indicate that both a trace and one of its suffixes are present in the set (as is the case for a and aa). Finally, one can also observe that more than one event can be appended to an existing trace at once, as is illustrated in the bottom-left of the last prefix tree.

A proxy π is said to be *combinatorial* if, for every $\overline{\sigma} \in \Sigma^*$ such that $|\overline{\sigma}| = n$, there exists a sequence of sets $S_j \subseteq \Sigma^*$ for $j \in [1, n]$ such that $\pi(\overline{\sigma})[...n] = S_1 \cdot \ldots \cdot S_n$. At each step of

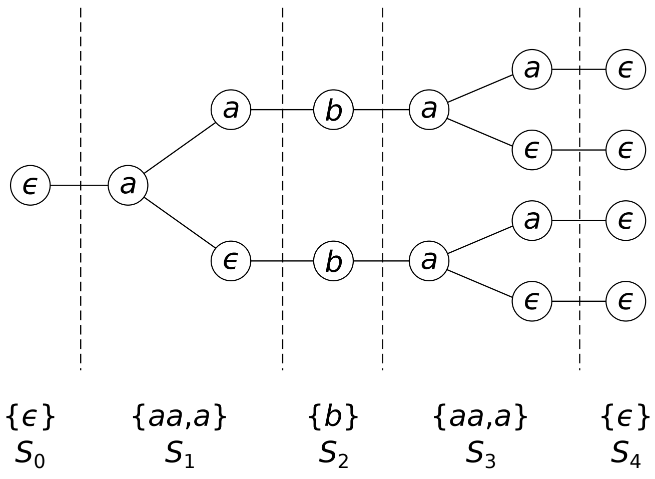


Figure 4.2 : Illustration of the operation of a combinatorial proxy.

its operation, a combinatorial proxy chooses a set of traces S_j, which are taken as the possible extensions of any trace in its current set. The proxy is called "combinatorial", as the traces it defines are the concatenation of any combination of traces picked from each S_j.

When described in terms of prefix trees, a combinatorial proxy is such that at any given step, it adds the same set of nodes to all the leaves of the current tree. For instance, the proxy of Figure 4.1 is *not* combinatorial, as, at each step, existing traces are not all extended in the same way. In contrast, Figure 4.2 shows the operation of a combinatorial proxy. The dashed lines delineate the portions of the prefix tree that are successively appended. Note how at each step, an identical node structure is appended to every leaf of the previous step. At the bottom of the figure, the sets of traces $S_1, \ldots, S_n$ corresponding to each structure are represented. One can check that every path in the tree is indeed the concatenation of a combination of a trace taken from each of the S_j.

Combinatorial proxies can be seen as independent of output history in that the possible ways in which an output trace extended are the same regardless of the content of that trace. Hence, in Figure 4.2, the set S_1 indicates that any possible trace produced so far can be extended either by one or by two a events. Although they can produce a strict subset of all behaviors available to an unrestricted proxy, in practice, almost every enforcement mechanism presented in past literature can be abstracted in the form of a combinatorial proxy, suggesting that this notion captures an important feature of enforcement.

In addition, some combinatorial proxies admit a natural representation under the form of an input—output automaton, such as a Mealy machine [137]. We recall that a Mealy machine is a variant of a finite-state automaton where transitions are labelled with an input symbol from an alphabet Σ_I and an output symbol from another alphabet Σ_O.

A Mealy machine can be defined as a 6-tuple $\langle S, S_0, \Sigma_I, \Sigma_O, T, G\rangle$ where: S is a finite set of states, S_0 is the initial state, Σ_I is a finite set of input alphabet, Σ_O is a finite set of output alphabet, $T : S \times \Sigma_I \to S$ is a transition function mapping pairs of a state and an input symbol to the corresponding next state, and $G : S \times \Sigma_I \to \Sigma_O$ is an output function mapping pairs of a state and an input symbol to the corresponding output symbol. The transition and output functions can coalesce into a single function $T : S \times \Sigma_I \to S \times \Sigma_O$. Note that we may overload the use of ϵ to designate the case where the proxy outputs nothing.

For a given event alphabet Σ, a combinatorial proxy can be specified as a special case of a Mealy machine where $\Sigma_I = \Sigma$ and $\Sigma_O = 2^{\Sigma^*}$. The input symbol corresponds to the input event given to the proxy, and the output "symbol" of the corresponding transition is the set of traces that extend each possible trace (i.e. one of the S_j in the definition above). Figure 4.3 gives an example of such a Mealy machine. We use $*$ on a transition to indicate any event not mentioned in another outgoing transition from the same state and the notation $*/\{*\}$ to indicate that an event should be output as is. Intuitively, this proxy outputs all input events without modification, except sequences of b where any b after the first may or may not be deleted from the output (represented by the fact that b and ϵ are the two possible extensions of

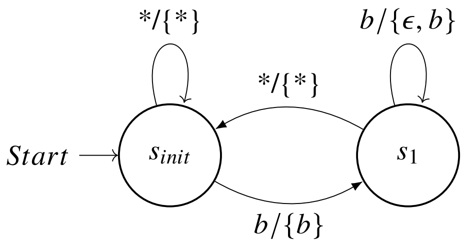


Figure 4.3 : A proxy suppressing successive b events following an initial b.

a trace in that case).

4.1.3 CATEGORIES OF PROXIES

The proxy is a high-level abstraction of any security mechanism that modifies the underlying execution to ensure compliance with the desired security policy. As such, it can best be seen as a component of an EM, as defined by Erlingsson [67]. Our previous definition is generic and gives the proxy almost unlimited freedom to modify the input trace it receives in various ways—even by outputting traces that are completely unrelated to the input. In practice, however, past literature has concentrated on specific types of proxies with stricter bounds on the modifications they are allowed to apply to a trace.

As detailed in Section 2.5.1, the main aspect that distinguishes different classes of EM monitors is their capacity to alter the input event stream. Early work on run-time enforcement usually distinguished between (1) an EM that is only capable of aborting (truncating) the execution; (2) an EM that can insert additional events in the input stream, (3) an EM can suppress (delete) events from the input stream and (4) an EM that can both insert and delete events [33]. This latter type is said to have the capacity to *edit* the input sequence.

Each of these alternatives corresponds to a different implementation strategy. For example, aspect-oriented programming [115] can be used to insert code segments that are

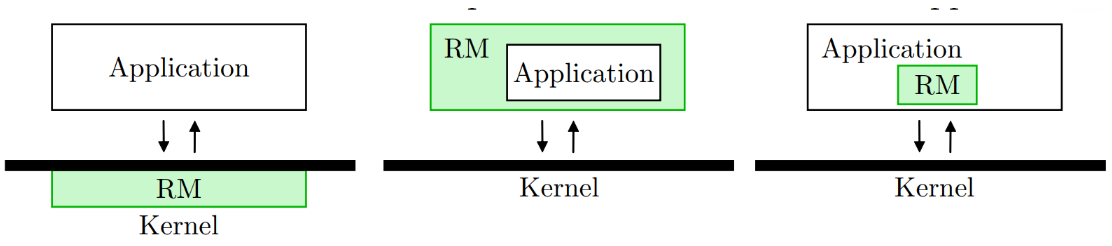


Figure 4.4 : The implementation of a monitor, from [67].

executed when different point cuts are encountered. This allows the implementation of an insertion proxy. Conversely, a firewall or an IDS interposed between the user and a host system operates as a suppression monitor since it can prevent service requests from being executed (in effect suppressing them). The heightened capabilities of an edition EM require the use of a more involved program-rewriting method.

As observed by Erlingsson [67], the monitor (termed reference monitor or RM) can also be inserted at different layers of the architecture, with a consequent impact on its ability to affect the execution. Erlingsson distinguishes three cases, illustrated in Figure 4.4. First, the monitor may operate inside the operating system's kernel space and prevent the execution of sensitive instructions (left). Alternatively, an untrusted program may be run in an interpreter, which simulates the execution and interposes itself between the program and the operating system (center). Finally, the monitor may be inlined inside of the target program, through a rewriting or code injection process (right). The execution of a program modified in such a manner can then be thought of as equivalent to the simultaneous execution of the program and the reference monitor.

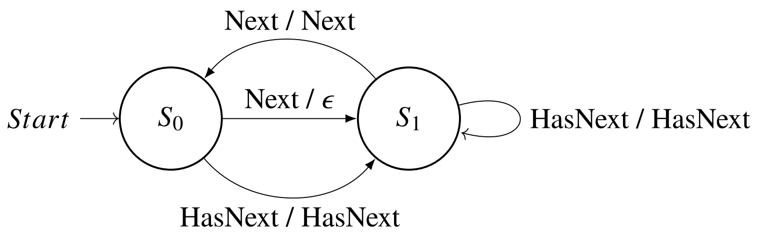


Figure 4.5 : A graphical representation of an EM of the Next and HasNext pattern policy.

4.1.4 EXAMPLES

We now refer to the policies we specified in Section 1.1.1.2 and talk briefly about how the enforcement mechanism can be used to enforce each policy:

For the Next and HasNext Pattern Policy, the policy is violated whenever a $Next$ event appears without being immediately preceded by a $HasNext$ event. To enforce the policy, possible actions can be done: inserting a $HasNext$ event, suppressing any $Next$ until a $HasNext$ event appears, or maintaining a buffer and storing all the $Next$ events until a $HasNext$ appears and then outputting all the events in the buffer. As an example, Figure 4.5 shows an EM that enforces the policy by suppressing any $Next$ event coming before a $HasNext$ event.

The mechanism used to enforce the fair dispatcher ordering policy [70] aims to reorder events. To do this, several buffers are used to store events temporarily to be later used in their correct order. To avoid buffering a large number of events, a possible buffer purging technique consists of deleting an event from the buffer, and a possible healing technique consists of adding an event into the trace that may correct the order without buffering the current event. Once reaching a specified threshold K_{heal} (indicating the number of events in the buffer after which healing is allowed) or K_{purge} (indicating the number of events in the buffer after which purging is allowed), healing or purging is used, respectively.

To enforce the privacy policies in OSNs [145], two tools are used to communicate with each other: An OSN with built-in enforcement for static privacy policies, and the LARVA tool [54] to monitor the evolution of the OSN and control the state of the policies at each moment in time.

For the file format policy enforcement, Pinisetty *et al.* [151] consider the case of predictive runtime enforcement and assume that the system is not entirely black-box, but they know something about its behavior based on prediction. The *a priori* knowledge of the system allows the EM to emit an output event instantly rather than delaying it until more events arrive, or even permanently blocking altogether.

4.2 PRODUCING CORRECTED TRACES

The previous two sections have presented the notions of policy and proxy in isolation. On one side, policies were expressed regardless of how they could concretely be enforced; on the other side, proxies were defined as entities that could alter an input trace but without any specific aim. In this section, we leverage these two concepts and describe a mechanism that involves an interplay between the output of a monitor for a given policy, and the alterations a proxy can inflict on an input trace to produce a trace that guarantees compliance with the policy (or more precisely, non-violation).

4.2.1 A DEFINITION OF CORRECTION

A proxy can be distinguished by whether it produces traces that are compliant or not with respect to a policy Φ. Formally, a proxy π is called *strongly* Φ*-preserving* if for any given input $\overline{\sigma} \in \Sigma^*$ such that $\pi(\overline{\sigma}) = \Sigma_1^*, \Sigma_2^*, \ldots, \Sigma_n^*$, any $i \in [1, n]$ and any $\overline{\sigma}' \in \Sigma_i^*$, we have that $\varphi(\overline{\sigma}') \in \{\top, \top^?\}$. The proxy strongly preserves Φ if it only outputs sequences that result in a positive verdict of the monitor induced by Φ. Similarly, π is *weakly Φ-preserving* if the latter condition is replaced by $\varphi(\overline{\sigma}') \in \{\top, \top^?, \bot_?\}$. The proxy weakly preserves Φ if it only outputs sequences that do not result in a definitive negative verdict with respect to Φ.

At first glance, it would seem to be sufficient to merely pipe an event trace into a Φ-preserving proxy π and pick any of its output traces as the corrected one. This solution discards two important elements. First, it assumes that π is Φ-preserving is a strong hypothesis, which couples the possible actions of a proxy with a specific policy. This means that a new proxy must be designed for every policy (or even every change of policy). In addition, even ensuring that a given π is Φ-preserving is a nontrivial task. To be convinced of this fact, consider the proxy that never makes any modification to an input trace and simply outputs it as is: demonstrating that this proxy is Φ-preserving amounts to solving the model checking problem $\pi \models \Phi$.

The second element to consider is the notion of transparency touched upon earlier. We recall that our definition of proxy can, in theory, correct an input trace in ways that are completely unrelated to the input, even for traces that are already in compliance with the policy. One must therefore ensure, at least, that prefixes of the input that do not violate the policy should be left as they are. But this leaves ambiguity as to the point at which a portion of the input should be replaced by a corrected version, and by how much. It also does not stipulate whether the proxy should keep on correcting the input forever after this moment or the contents of the input trace could be used again after some time, and from what point.

In the remainder of this section, we clarify these questions by proposing a formal definition of how a proxy π is expected to interact with an input trace, with respect to an independently specified policy Φ. We do so by considering an "original" input trace $\overline{\sigma} \in \Sigma^*$ and a "corrected" output trace $\overline{\sigma}' \in \Sigma^*$, and we give conditions as to how these two traces

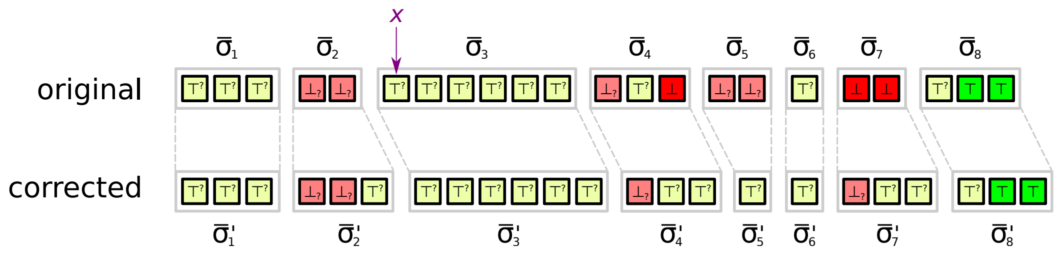


Figure 4.6 : Illustration of the relationship between an input trace and a possible correction.

should be related.

Given a trace $\overline{\sigma}$ and a suffix $\overline{\sigma}'$ to be appended to it, we say that $\overline{\sigma}'$ is *positive throughout* if $\varphi(\overline{\sigma} \cdot \overline{\sigma}'') \in \{\top, \top^?\}$ for every $\overline{\sigma}'' \preceq \overline{\sigma}'$. Intuitively, the suffix is positive throughout if appending each event successively always results in the monitor producing a positive verdict. We say that $\overline{\sigma'}$ *starts negatively* if $\varphi(\overline{\sigma} \cdot \overline{\sigma}'[0]) \in \{\bot_?, \bot\}$. The suffix *starts negatively* if appending its first event to $\overline{\sigma}$ produces a negative verdict. Finally, the suffix *ends positively* if $\varphi(\overline{\sigma} \cdot \overline{\sigma}') \in \{\top, \top^?\}$ –that is, if the monitor's verdict after appending all its events is positive.

We divide $\overline{\sigma}$ and $\overline{\sigma}'$ into n *segments* such that $\overline{\sigma} = \overline{\sigma}_1 \cdot \ldots \cdot \overline{\sigma}_n$, and $\overline{\sigma}' = \overline{\sigma}'_1 \cdot \ldots \cdot \overline{\sigma}'_n$. The corresponding segments of each trace do not need to be of the same length. We call such a division of the original and corrected trace a *segmentation*. We designate by *corrected prefix up to i* the prefix of the corrected trace containing the first i segments (this prefix being set to ϵ for $i = 0$). A segmentation of $\overline{\sigma}$ and $\overline{\sigma}'$ is called a *correction* if, for every corrected prefix up to i ($i \in [0, n]$), $\overline{\sigma}_{i+1}$ is positive throughout and $\overline{\sigma}'_{i+1} = \overline{\sigma}_{i+1}$, or $\overline{\sigma}_{i+1}$ starts negatively and $\overline{\sigma}'_{i+1}$ ends positively.

Figure 4.6 shows an example of such a correction. The squares at the top represent the events of the original trace and those at the bottom the events of the corrected trace. Each of these events is grouped according to a possible segmentation, with the boundaries of the corresponding segments in the original and corrected trace linked by dashed lines. The colour and symbol inside an event of a segment σ_i correspond to the verdict the monitor reaches when ingesting the corrected segment up to $i-1$, plus the events in the segment up to this point. For instance, the verdict associated to the event x in the figure is obtained by feeding the monitor the first two segments of the corrected trace and then event x of the original trace, i.e. $\varphi(\overline{\sigma}'_1 \cdot \overline{\sigma}'_2 \cdot x)$.

The diagram shows the various situations covered by the definition. Segment $\overline{\sigma}_1$ is positive throughout; therefore, segment $\overline{\sigma}'_1$ is identical. This corresponds to the transparency requirement, which imposes that a prefix of the original trace that satisfies the policy should be left untouched. Segment $\overline{\sigma}_2$ starts negatively. This gives the signal that this event and whatever comes after can be substituted by another segment, $\overline{\sigma}'_2$. In the example, observe that the corrected segment contains more events than the input segment it is matched with. Also observe that, as per the definitions above, the corrected segment must end positively. Thus, a transducer producing a corrected trace can replace a segment of arbitrary length that starts negatively by another segment, but this new segment must be such that the resulting trace satisfies the policy. That is, it cannot emit a correction that places the trace on a "cliffhanger" where it still does not satisfy the policy. However, note how $\overline{\sigma}'_2$ contains events resulting in a negative verdict. The only requirement is that the policy is satisfied at the end of the prefix.

On its side, $\overline{\sigma}_3$ is positive throughout, and thus must be output identically as $\overline{\sigma}'_3$. This is an important aspect of our definition of correction: once a segment has been corrected and the trace has been put into a satisfying state, events of the input trace from this point on must resume being outputted without modification as long as the trace satisfies the policy. This condition is stricter than existing definitions of transparency, which only impose that a prefix of the original trace should be let through until the first violating event. That is, once the input trace violates the policy, the classical definition of enforcement allows a proxy to alter the trace and does not rule out that it can do so forever. In contrast, our definition of correction stipulates that, once a violating input segment has been replaced by a fixed version, control must be relinquished to the original input trace until a new violation is found.

4.2.2 SELECTING CORRECTIONS

Referring back to the definition of *enforcement* as defined in [33], we see that a monitor is considered to have enforced a property if it can transform an invalid sequence into *any* valid sequence. A transparency requirement is present in the definition, but it only limits the transformations that the monitor may perform on valid traces. Being consistent with this definition, the monitor may be able to enforce a property, but not in a way that is necessarily useful or desirable. Indeed, few people would accept a security mechanism that “corrects” a misbehaving execution by replacing it with a benign input that is completely unrelated to the original execution.

Consequently, additional constraints must be imposed on the behavior of the monitor in order to ensure meaningful enforcement. For example, Bielova *et al.* [37] define a series of monitors whose output is syntactically related to input sequence—for instance imposing that the corrected sequence always be a prefix of the input sequence. Alternatively, Khoury *et al.* [113] suggest that all possible sequences be arranged in a preorder and that the monitor be required to select a solution that falls higher than the input on this preorder. In this thesis, we adopt a similar, but more flexible solution: by separating the generation of potential solutions from the selection of the optimal (or preferred) solutions, we do away with the complexity of creating such a preorder and ensuring that the monitor behaves in a manner that is conformant with this added restriction.

A *scoring function* is a function $\rho : \Sigma^* \to \mathbb{R}$, which assigns to each sequence in Σ^* a real value called its *score*. This function induces a total ordering $\sqsubseteq$ on traces, such that $\overline{\sigma} \sqsubseteq \overline{\sigma}'$

if and only if $\rho(\overline{\sigma}) \leq \rho(\overline{\sigma}')$. In principle, any preorder can be used to select the optimal corrected sequence. In practice, some preorders can exhibit properties that may be desirable. First, a preorder can be *monotonic*, meaning that a corrective action taken by the monitor can never irremediably 'wreck' a sequence in the sense that a continuation of the original sequence falls higher on the preorder that any possible corrected sequence. Formally:

$$\forall \overline{\tau}, \overline{\tau}' \in \Sigma^* : \overline{\tau} \sqsubseteq \overline{\tau}' \Rightarrow \neg\exists\overline{\sigma} \in \Sigma^\omega : \forall\overline{\sigma}' \in \Sigma^\omega : \overline{\tau}' \cdot \sigma' \sqsubseteq \overline{\tau} \cdot \overline{\sigma}$$

Second, a preorder can be *truth-correlated*, meaning that any valid trace has a score higher than any invalid trace. Formally, this means that for a given policy φ we have that $\forall \overline{\tau}, \overline{\tau}' \in \Sigma^\omega : \varphi(\overline{\tau}) \wedge \neg\varphi(\overline{\tau}') \Rightarrow \overline{\tau}' \sqsupseteq \overline{\tau}$.

A natural example of choice for the preferred trace could minimize the number of modifications (insertions and deletions) on the input. This can be captured by turning a given input alphabet Σ into an alternate alphabet $\widehat{\Sigma}$, where each symbol $\sigma \in \Sigma$ exists in three versions: σ designates an event of an output trace that was already present in the input, $\sigma_\downarrow$ designates an event that was added to the output, and $\sigma^\uparrow$ designates an event that was deleted from the input. For example, the trace abc, to which b is deleted and a is inserted at the end, would result in the trace $ab^\uparrow ca_\downarrow$. Evaluating a policy on such a trace can be done by simply handling any $\sigma_\downarrow$ as σ, and any $\sigma^\uparrow$ as ϵ. Thus, the previous trace would be handled by a policy (and its associated monitor) in the same way as aca.

Equipped with this notation, defining a scoring function that correlates with modifications is straightforward: for a given trace $\overline{\sigma}$, one simply adds -1 for each occurrence of an inserted or deleted event (starting from 0). Thus, a higher-ranking value corresponds to a corrected trace with fewer modifications. However, this is by far not the only possible ranking one can build. As another example, one could imagine a function that assigns a higher score to a sequence that satisfies the policy for as many prefixes as possible. Concretely, this could be defined as $\rho(\overline{\sigma}) \triangleq |\{\overline{\sigma}' \leq \overline{\sigma} : \mu(\overline{\sigma}') \in \{\top^?, \top\}\}|$. We perform some experiments in Section 5.3 that describe other scoring functions specific to some use cases.

4.3 A MODULAR RUNTIME ENFORCEMENT PIPELINE

Equipped with the notions of monitor, proxy and the concept of corrected trace, we now present an alternate model of runtime enforcement to transform the input sequence in order to ensure both the respect of the security policy as well as provide assurance that the corrected sequence is optimal with respect to a separate transparency requirement. The key idea of this model is to separate the various operations of enforcement into independent computation steps.

4.3.1 PIPELINE DESCRIPTION

The proposed enforcement model takes the form of a "pipeline", which is a chain of transducers taking as its input a possibly incorrect event trace and producing as its output a sequence of events that satisfies the definition of correction with respect to a policy introduced in Section 4.2.1. The high-level schematics of the model are shown in Figure 4.7. Various transducers are represented as boxes illustrated with different pictograms, depending on their definition. These transducers are organized along a data flow graph where events move from left to right. A link between two transducers indicates that the output of the first is given as the input to the second.

The diagram uses different colours to represent events of different types, which will be explained later. The pipeline is parameterized by three transducers, labeled μ, π, and ρ. First,

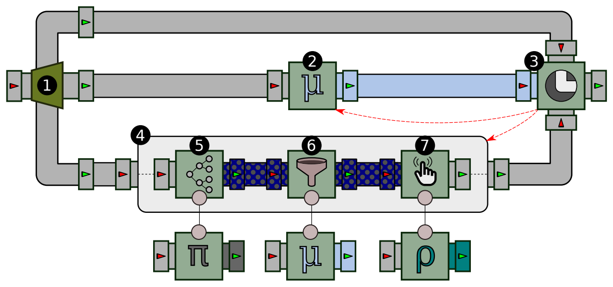


Figure 4.7 : The stages of the runtime enforcement framework. Events flow from left to right.

μ is a *monitor* responsible for evaluating a security policy on a trace. Transducer π is the *proxy*, which is tasked with applying modifications to an input trace. Finally, ρ is the *ranking* transducer that assigns a numerical score to a trace based on an enforcement preorder.

The global operation of the pipeline can be summarized as follows. An input event sequence is first forked into three separate copies, as represented by box #1 in the figure. One copy is fed to an instance of the monitor μ (box #2). Another copy is fed to an enforcement pipeline (box #4), itself decomposed into three phases. First, a single event sequence is turned into multiple event sequences by applying the possible corrective actions produced by a proxy transducer π (#5); this set of sequences is then filtered out so that only sequences satisfying the security policy evaluated by μ are kept (#6). The last phase sends the remaining sequences into the ranking transducer ρ, and picks the one with the highest rank as specified by the enforcement preorder (#7).

The last step of the pipeline is represented by box #3, which is called a *gate*. Based on the output from the monitor (box #2), the gate either outputs elements of the original trace directly (if it is valid) or switches to the output from the enforcement pipeline emitting a corrected sequence. Depending on the actual sequence of events produced by the gate, the internal state of the upstream transducers may need to be forcibly updated; this process, called *checkpointing*, is represented by the backward red arrows. In the remainder of this section, we

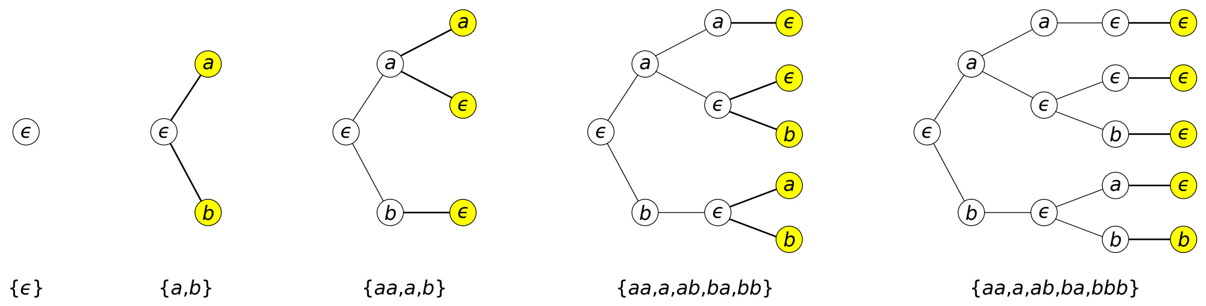


Figure 4.8 : A modified sequence of prefix trees where only leaves are modified, and where a single event at a time is appended to each branch. This sequence ends up producing the same set of traces as that of Figure 4.1.

describe the stages of this pipeline in more detail and end with a discussion of the advantages of this model.

4.3.1.1 PRODUCTION OF CORRECTED TRACES

In our earlier definition of a proxy, each output multi-event contains the complete set of sequences proposed in replacement of the input. This introduces a large amount of repetition since a prefix of each of these sequences was already present in the set produced for the previous event. It also prevents the output of the proxy from being ingested by another downstream transducer in a progressive manner. Better yet would be a representation which, upon each input event, only produces a description of what is appended to the existing prefix tree. To this end, we introduce a further restriction on the sequence of prefix trees induced by a proxy, by imposing that each leaf of a given tree be appended by at least one node in the next tree and that only single nodes can be added at each step (instead of sequences of nodes).

One can reason that a proxy following this constraint can produce the same set of sequences as an unconstrained one; Figure 4.8 illustrates this. It shows a sequence of prefix trees following the restriction added in this section. It can be observed that the final prefix tree represents the same set of traces as that of Figure 4.1. However, this is obtained at the price of additional ϵ nodes due to the condition that each leaf must be appended with at least one node at every step. In addition, one more step in the sequence is required to append the two b events at the bottom of the tree that was added in a single step in Figure 4.1.

The restriction adds some complexity to the trees; in contrast, they give a regular structure to these trees where each branch has the same length and where each further step adds a single layer of nodes to the leaves of the current tree. In counterpart, this regularity can be exploited; for the purpose of the enforcement pipeline, a special representation of these trees has been adopted such that their contents can be transmitted in the form of a sequence of events. Let $\mathcal{V}\langle T]$ denote the set of vectors of elements in T. For a given vector $v \in \mathcal{V}\langle T]$, let $v[i]$ denote the element at position i in that vector. Define $\mathcal{T} = \mathcal{V}\langle\mathcal{V}\langle\Sigma]]$ as the set of prefix tree elements, which are vectors of vectors of events. A prefix tree sequence is a trace $v_0, v_1, \ldots, v_n \in \mathcal{T}^*$, such that $v_0 = \langle[\ ]]$, and for each $i \in [1, n]$:

$$|v_i| = \sum_{j=0}^{|v_{i-1}|} |v_{i-1}[j]|$$

The intuition behind this condition is that the j-th vector within a prefix tree element corresponds to the list of children attached to the j-th symbol in the prefix tree element that precedes it. As an example, the prefix tree sequence in Figure 4.8 corresponds to the sequence of prefix tree elements:

$$\langle [\ ] \rangle$$
$$\langle [a], [b] \rangle$$
$$\langle [a, \epsilon], [\epsilon] \rangle$$
$$\langle [\epsilon], [\epsilon, b], [a, b] \rangle$$
$$\langle [\epsilon], [\epsilon], [\epsilon], [\epsilon], [b] \rangle$$

Note that a symbol may be ϵ, so a tree of a given depth does not necessarily represent sequences of equal lengths. This representation makes it possible for a transducer to output a sequence of elements that represents the progressive construction of a prefix tree representing multiple event sequences. The task of box #5 in Figure 4.7 is precisely to receive each sequence set produced from π, and turn it into the appropriate prefix tree element.

4.3.1.2 FILTERING OF VALID TRACES

The purpose of this setup becomes apparent in the next phase of the enforcement pipeline. The set of event traces generated by the proxy captures all the possible replacements of the original input trace. However, some of them are valid according to a given security policy, and others are not; one must therefore remove from the possible sequences produced by the proxy all those that violate the policy.

The task of filtering invalid traces is represented by box #6 in the pipeline. It receives as input a sequence of prefix tree elements and produces as output a modified sequence of prefix tree elements, where any branches corresponding to prefixes violating the security policy are pruned out. If the monitor produces $\perp$ anywhere along a path, the node producing this verdict and all its descendants in the prefix tree are replaced by a placeholder $\diamond$, indicating that these nodes should not be considered. If a path ends with the monitor producing $\perp^?$, the last node of that path is replaced by $\diamond$. For example, suppose that the security policy imposes that a trace never starts with b. In the rightmost tree of Figure 4.8, the first b node under the root must therefore be deleted. In this particular case, the output of the filtering step would be the sequence of prefix tree elements $[[]], [[a, \diamond]], [[a, []], [\diamond]], [[[]], [[], b], [\diamond, \diamond]], [[[]], [[]], [[]], [\diamond], [\diamond]]$.

Conceptually, it suffices to run a fresh instance of μ on each path of the induced prefix tree and to remove a node (as well as all its descendants) as soon as μ appends $\perp$ to its output. However, the process needs to be done incrementally since the contents of the prefix tree are produced one element at a time. Algorithm 2 shows how this can be done. The algorithm receives a vector of monitor instances and a prefix tree element of the same size. The μ_{σ_i} represents the state of monitor μ after processing the paths ending in each leaf of the prefix tree, and the v_i are the children events to be appended to each of these leaves. For each μ_{σ_i} and v_i, the algorithm iterates over each event x in v_i and adds to an output vector m the monitor instance $\mu_{\sigma_i \cdot x}$, which is the result of feeding x to μ_{σ_i}. If the resulting output trace contains $\perp$, this path violates the security policy and the event x is replaced by $\diamond$. Otherwise, the event is added to the output vector, and the process repeats. The end result is a new pair of vectors m and v, where v is the filtered prefix tree element obtained from $[v_0, \ldots, v_n]$ and m is the vector of monitor states for each leaf of this element.

As with the previous step, note that this operation is independent of the formal notation used to represent the security policy. It is applicable as long as the monitor is a computational entity outputting a sequence of elements in $\mathbb{B}_4$ and that stateful copies of itself can be cheaply produced.

Algorithm 2 Incremental filtering

```
procedure $\textsc{Filter}([\mu_{\sigma_1}, \ldots, \mu_{\sigma_n}], [v_0, \ldots, v_n])$
    $v \leftarrow [\,], m \leftarrow [\,]$
    for $i \leftarrow 1, n$ do
        $v' \leftarrow [\,]$
        for $x \in v_i$ do
            $\textsc{Add}(m, \mu_{\sigma_i \cdot x})$
            if $\mu_{\sigma_i}(x)$ contains $\perp$ then
                $\textsc{Add}(v', \diamond)$
            else if $i = n$ and $\mu_{\sigma_i}(x)$ ends with $\perp^?$
                $\textsc{Add}(v', \diamond)$ else $\textsc{Add}(v, v')$
            end if
        end for
    end for
    return $(m, v)$
end procedure
```

Algorithm 3 Output trace selection

```
procedure $\textsc{Update}([(\rho_{\sigma_1}, s_1), \ldots, (\rho_{\sigma_n}, s_n)], [v_0, \ldots, v_n])$
    $m \leftarrow [\,]$
    for $i \leftarrow 1, n$ do
        for $x \in v_i$ do
            $s = -\infty$
            if $x \neq \diamond$ then
                $s \leftarrow \textsc{Last}(\rho_{\sigma_i}(x))$
            end if
            $\textsc{Add}(m, \rho_{\sigma_i \cdot x})$
        end for
    end for
    return $m$
end procedure
```

4.3.1.3 SELECTION OF THE OPTIMAL OUTPUT TRACE

This final phase of the enforcement pipeline relies upon a special transducer, called the *selector*, which receives as input a sequence of prefix tree elements, and attempts to select the "optimal" one, based on a transparency condition. This phase involves the ranking transducer $\rho : \Sigma^* \rightarrow \mathbb{R}$, which assigns a numerical score to a trace. The principle of the selector is simple: each path in the filtered prefix tree is evaluated by ρ, and the path that maximizes the score is selected and returned as the output.

The operation of the selector, depicted in Figure 4.7 as box #7, is described by procedure Update in Algorithm 3. This time, the procedure receives a prefix tree element $[v_0, \ldots, v_n]$ and a vector of pairs, each containing a ranking transducer instance ρ_{σ_i} and the score s this transducer has produced after processing σ_i. The algorithm then proceeds in a similar way as for Filter: each transducer instance is fed with each child in sequence, and the updated instance and its associated score are added to the new vector m. Applying this procedure successively on each prefix tree element, and feeding the output vector m back into the next call to Update produces a vector, from which the output trace σ_i can be chosen based on the highest score s_i in all pairs.

4.3.1.4 MERGING VALID VS. CORRECTED TRACES

The last step of the pipeline, called the *gate* and represented by box #3, takes care of letting the input trace through as long as it does not violate the security policy, and it switches to the output of the enforcement pipeline only in case of a violation. This is why the gate receives as its inputs the original event trace, the output from the enforcement pipeline, as well as the verdict of the monitor μ for events of the input trace (box #2) that allows it to switch between the two. More precisely, the gate returns an input event directly if and only if μ does not produce the verdict $\perp$ or $\perp^?$ upon receiving this event. Otherwise, this event is kept in an internal buffer, and the gate waits for an event or a sequence of events to be returned by the enforcement pipeline of box #4, which is output instead. As long as μ returns a false or possibly false verdict, input events are added to the buffer and also fed to the enforcement pipeline. In such a way, the enforcement pipeline is allowed to ingest multiple input events and replace them with another sequence.

This mode of operation ends at the earliest occurrence of two possible situations. The first is if the monitor resumes returning either $\top$ or $\top^?$. In such a case, the input events in the buffer are deemed to be a safe extension of the ongoing trace and are sent to the output. The second situation is if the enforcement pipeline produces a corrective sequence as its output. This indicates that the sequence of buffered input events must be discarded, and replaced by the output of the enforcement pipeline. After either of these two situations occur, the input buffer is cleared, and control is returned to the input trace.

However, doing so requires a form of feedback from the downstream gate to the upstream transducers so that their internal state is consistent with the trace that has actually been output and not the input trace that has been observed. To illustrate this notion, consider a simple security property stating that every a event must be followed by a b. If the input trace is ac, the first a event is output directly, as this prefix does not violate the policy. The next event, c, makes the prefix violate the policy; the gate therefore switches to the output of the enforcement pipeline. Suppose that this pipeline produces as its output the corrective sequence bc, which inserts a b before the c. This sequence restores compliance with the policy, and events from the input trace can again be let through. However, the monitor μ of box #2, in charge of evaluating compliance of the trace, is still in an error state (having read ac); its verdict will therefore be incorrect for the subsequent incoming events.

This entails that one must be able to "rewind" μ and put it in the state it should be after reading the real output trace (abc) so that it produces the correct verdict for the next events. It is the purpose of the feedback mechanism illustrated by the red arrows in Figure 4.7, which we call *checkpointing*. Along with the transducer μ of box #2, a copy μ_σ is kept of that transducer in the state it was after reading σ (the "checkpoint"). Intuitively, σ represents the sequence of events that has actually been output by the pipeline. As events are received, μ updates its internal state accordingly, but μ_σ is preserved. This copy is updated only when the downstream gate instructs it to be updated, by providing a segment of newly output events σ'. When this occurs, both the checkpoint μ_σ and the internal state of μ are replaced by $\mu_{\sigma\cdot\sigma'}$. A similar feedback process occurs for the enforcement pipeline of box #4.

On its side, the gate notifies these transducers of a new checkpoint every time it outputs an event from the original input trace, or when a corrected segment from the enforcement pipeline is chosen instead. This ensures that the whole system is always in sync with the contents of the actual output sequence.

4.3.1.5 EVENT BUFFERING

A final aspect of the architecture that needs to be discussed is the notion of buffering. The default behavior of the selector (box #7) is to keep accumulating prefix tree elements without producing output until a signal to pick a trace is given to it. This makes it possible to consider corrective actions generated by the proxy that may involve replacing a sequence of input events with another sequence of output events. However, the question remains as to how and when this signal should be emitted. The definition of a corrected trace in Section 4.2.1 and the proposed architecture both deliberately leave this parameter open, enabling a user to select among various possibilities. We enumerate a few of them in the following.

The first is a greedy choice: every time the selector receives a prefix tree element, it picks the event that maximizes the evaluation of the ranking transducer (evaluated from the beginning of the trace) and immediately outputs it. This greedy strategy does not guarantee the absolute best course of action unless the scoring transducer is *suffix-monotonic*. Formally, a transducer τ is suffix-monotonic if for every triplet of sequences σ_1, σ_2, and σ_3, the fact that $\tau(\sigma_1) \leq \tau(\sigma_2)$ implies that $\tau(\sigma_1 \cdot \sigma_3) \leq \tau(\sigma_2 \cdot \sigma_3)$. In such a case, one can easily see that picking the best choice at every event guarantees the best score overall.

The second strategy is to pick an output trace once a given threshold length is observed. Prefix tree elements are buffered until k is received, after which the best path in the tree is selected (note that this path itself may be shorter than k due to the presence of ϵ symbols). Yet another possibility is to buffer events until one of the traces reaches a threshold score. Finally, one last possibility is to base the decision to pick a trace on a condition evaluated on the prefix tree itself — for example, by evaluating an auxiliary monitor $\delta : \Sigma^* \rightarrow \mathbb{B}_4$ on each path. As an example, one could decide to pick a trace whenever a specific event is observed in one of the paths. Obviously, the appropriate choice is specific to the use case and the nature of the properties involved in the enforcement pipeline.

The pipeline as defined ensures transparency as it is usually defined. A more conservative strategy would be to only alter the execution if the input sequence irremediably violates the security policy. This strategy guarantees compliance with transparency as it is usually defined but limits the monitor to the enforcement of safety properties since a violation of a liveness property can always be remediated by subsequent actions. However, one may instead opt for a more flexible notion of transparency, which allows modifications of valid traces, as long as the output is guaranteed to be higher than the input on the enforcement preorder. In many cases, the strategy employed will be context-specific, imposing that some element of the input be preserved or obligating the selector to take action once a specific event is encountered in the input trace. This would likely be the case for most transactional properties.

Regardless of the strategy chosen, one should keep in mind that the notion of an “optimal” output sequence must be qualified with respect to the choices available to the selector at the moment an output must be produced.

4.3.2 USE CASES

We now illustrate the operation of this pipeline by describing two use cases, on which we completely define all elements of the workflow.

4.3.2.1 USE CASE 1: MUSEUM

In the first use case, we look into an example taken from Drabik *et al.* [64], which considers two sorts of visitors entering a museum: children and adults, and guards responsible for protecting the visitors.

We are interested in a more complex scenario by studying the behavior of the principals while both entering and leaving the museum. The set of events that can occur in the trace of a museum: g^+, c^+, and a^+ indicating a guard, a child and an adult entering the museum, respectively, and g^-, c^-, and a^- indicating a guard, a child and an adult leaving the museum, respectively.

Monitor To keep the children safe while they are inside the museum, a policy stating that access is forbidden for any child unless there are at least as many guards as children in the museum should be enforced. Adults are allowed to enter on their own; however, children must be accompanied by a guard. The property involves keeping track of the number of children inside the museum as it increases and decreases over c^+ and c^- events that may occur, as well as the number of guards that changes over the g^+ and g^- events. There are multiple ways this policy can be stated, but a particularly appropriate notation is through a system of stream equations over typed stream variables as defined in Lola (described in Section 1.4). A stream expression may involve the value of a previously defined stream.

The language provides the expression $\text{ite}(b; s_1; s_2)$, which represents an if-then-else construct: the value returned depends on whether the predicate of the first operand evaluates to true. It also allows a stream to be defined by referring to the value of an event in another stream k positions behind, using the construct $s[-k, x]$, where s is a stream name, k is the offset, and x is a possible value in stream s. If $-k$ corresponds to an offset beyond the start of s, a constant value x is used instead. Finally, two streams can be combined to form another stream; hence $s_1 + s_2$ designates the stream where each event is obtained by taking the sum of events at the corresponding position in both s_1 and s_2.

$$t_1 := \text{ite}(g^+; t_1[-1,0] + 1; \text{ite}(g^-; t_1[-1,0] - 1; t_1[-1,0]))$$
$$t_2 := \text{ite}(c^+; t_2[-1,0] + 1; \text{ite}(c^-; t_2[-1,0] - 1; t_2[-1,0]))$$
$$\varphi := \text{ite}(\varphi[-1,\top], (t_1 - t_2) \geq 0, \bot)$$

The first equation defines a stream that keeps the count of the number of guards inside the museum. This counter is incremented by one whenever a guard enters the museum, decremented by one whenever a guard leaves the museum and keeps its value otherwise. The second equation does the same thing for children. We assume that $t_1 = 0$ and $t_2 = 0$ whenever an expression refers to a position before the start of the stream. The third equation represents the idea that the number of children inside the museum should never exceed the number of guards. A Boolean stream is defined as φ, the output of which can be used as the monitor verdict for the security policy. The equation is made such that the property remains false once

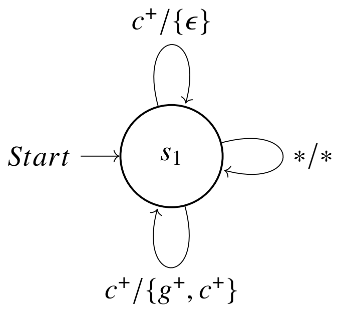


Figure 4.9 : Representation of a possible proxy enforcing the museum use case.

it becomes false.

Proxy The interest of this scenario lies in the possible variations for the proxy and enforcement preorder. The policy can be enforced by refusing (suppressing) the entrance of a child when the number of guards inside the museum is zero or by lending (inserting) a guard when needed.

One may also buffer all c^+ events until a c^- event appears. Then we can output one c^+ event from the buffer or until a g^+ event appears. Then we can output all the c^+ events from the buffer.

In Figure 4.9, we provide an example of a proxy for the museum example defined as a Mealy machine. This proxy may either suppress a c^+ or insert a g^+ event.

Selector This policy exposes itself to several interrelated courses of action, with the choices made by the monitor restricting its future course of action: refusing a high number of children incurs its own trade-off since the museum will lose an amount of profit that it may gain if the children are allowed to enter. Similarly, adding more guards also incurs a trade-off because the museum must afford the salaries paid to these guards. The enforcement pipeline will be forced to choose between these courses of action in order to attain one of several goals. This time, we opt for TK-LTL described in Section 1.4.

The possible enforcement preorders in the museum use case can be to minimize the number of modifications to the trace, to maximize the number of children that enter the museum or to minimize the number of time steps where guards are “idle” (present while no children are there).

The process of expressing the enforcement preorder is straightforward, and most of the possible requirements can be formulated as relatively simple formulas. For instance, the TK-LTL sub-formula $\widehat{C}^{\top}_{g^+}$ counts the total number of guards that enter the museum, and the TK-LTL sub-formula $\widehat{C}^{\top}_{g^-}$ counts the total number of guards that leave the museum. Hence, the formula $\widehat{C}^{\top}_{g^+} - \widehat{C}^{\top}_{g^-}$ can be used as a transparency constraint if the museum's main concern is to minimize the total number of “idle” guards that are inside the museum. A monitor that seeks to achieve this goal will thus avoid inserting g^+ events in the input stream.

Similarly, the formula $\widehat{C}^{\top}_{c^+} - \widehat{C}^{\top}_{c^-}$ expresses an alternative transparency requirement, namely maximizing gains for the museum by allowing the highest number of visitors to enter, including the children.

4.3.2.2 USE CASE 2: CASINO

As a more complex example, we now consider a variant of the scenario from Colombo *et al.* [56], which stems from a study of the remedial actions that can be taken to recover from violations of the terms of smart contracts. Since smart contracts are transactional in nature and cannot be modified after they are deployed, the framework proposed in this thesis is especially well-suited to this situation.

In this example, the security policy dictates the interaction between three types of principals: the casino, players and dealers. The casino provides a venue where dealers can set up games of chance. In [56], it is suggested that such a game could be implemented by having the dealer generate a random number, publishing a hash of this number and asking players to guess the parity of the number that the dealer has generated. A player who wishes to participate joins by depositing a participation fee in the bank's account and submitting their guess. After a pre-specified time has elapsed, the dealer reveals the result and pays out to the winners. A player who correctly guessed the parity of the number gets back twice their participation fee, paid by the dealer. If a player loses, they forfeit their participation fee, which is divided equally between the dealer and the casino.

The following set of events can occur in a trace of the casino: $NewGame(A)$ indicates the onset of a game by dealer A, while $Bet(A)$ indicates that player A has placed a bet. The occurrence of the $EndGame()$ event indicates the end of the game and enjoins the selector to cease buffering events and take corrective action if needed. A payment from A to B will be noted by the event $Pay(A, B)$. All bets are worth two dollars (players who wish to bet more simply output multiple bet events), and the $Pay()$ event transfers a single dollar. We omit from events any element of a parameter value that does not bear consequence on the discussion of the event at hand. For instance, it is safe to assume that the $EndGame()$ action indicates the id of the game that must be ended, but we need not concern ourselves with these implementation details. We write $Bet(\cdot)$ as a shorthand for $\bigvee_x Bet(x)$, for any players x in the game. We likewise write $Pay(A, \cdot)$ (resp. $Pay(\cdot, A)$) for any payment in which principal A is the recipient (resp. donor).

Monitor The policy that underpins this scenario is as follows: while a game is in progress, the balance of the dealer's account can never fall below the sum of the expected payouts. This property involves keeping track of the dealer's balance as it increases and decreases over Pay

events that may occur.

Defining the security policy using LOLA becomes straightforward. The original event stream of casino events is first pre-processed to produce the Boolean streams e, b, p^+, and p^-, indicating whether an event is respectively an $EndGame$, a bet placed by a player, a payment from the player to the casino, or the reverse situation.

$$\begin{aligned}
t_1 &:= \mathrm{ite}(e; 0; \mathrm{ite}(b; t_1[-1, 0] + 2; t_1[-1, 0]))\\
t_2 &:= \mathrm{ite}(p^+; t_2[-1, k] + 1; \mathrm{ite}(p^-; t_2[-1, k] - 1; t_2[-1, k]))\\
\varphi &:= \mathrm{ite}(\varphi[-1, \top]; (t_2 - t_1) \geq 0, \bot)
\end{aligned}$$

The first equation defines a stream that keeps count of the potential payouts to players. This counter is reset to zero whenever a game ends; otherwise, it is incremented by two whenever a player places a bet and keeps its value otherwise. The second equation keeps track of the dealer's balance, assuming the trace starts with an initial balance k. It increments by one when a player pays the casino, and decrements by that same amount in the reverse situation. Otherwise, the balance is left unchanged. We assume that $t_1 = 0$ and $t_2 = k$ whenever an expression refers to a position before the start of the stream, where k represents the dealer's initial balance. The third equation represents the idea that the potential payouts should never exceed the current balance. A Boolean stream is defined as φ, the output of which can be used as the monitor verdict for the security policy. The equation is made such that the property remains false once it becomes false.

Proxy The policy can be enforced by refusing (suppressing) bets when the dealer's assets are insufficient to cover them or by lending (inserting) funds to the dealer's account. If a dealer is running multiple games simultaneously, the casino may also enforce the policy by prematurely

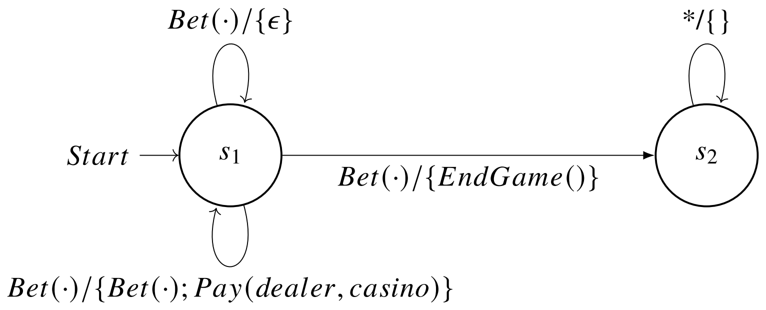

$$\begin{tikzcd}
\text{Start} \arrow[r] & s_1 \arrow[loop above, "Bet(\cdot)/\{\epsilon\}"] \arrow[loop below, "Bet(\cdot)/\{Bet(\cdot);Pay(dealer,casino)\}"] \arrow[rr, "Bet(\cdot)/\{EndGame()\}"'] & & s_2 \arrow[loop above, "*/\{\}"]
\end{tikzcd}$$

Figure 4.10 : Representation of a possible proxy enforcing the casino use case.

ending some games, in the hopes that the winnings incurred by the dealer may allow them to accept further bets on other games. Refusing the bets submitted by a player incurs its own trade-off since a player whose bets are consistently rejected may eventually take their business to a competing casino. Several formalisms can be used to represent the proxy. Furthermore, since the proxy is stated independently of the monitor and the selector, a different formalism can be used to represent each. In Figure 4.10, we provide an example of a proxy for the casino example defined as a Mealy machine. This proxy may either suppress a bet or terminate the game (by inserting an $EndGame()$ event).

Other examples of constraints that could be enforced on the the behavior of the proxy include: the proxy should not end 2 games successively (there should not be 2 consecutive $EndGame()$ events), the proxy should not refuse more than 5 consecutive bets (there should not be 5 successive ϵ), the proxy cannot do the three possible actions successively (the events $\{EndGame(), \epsilon, Pay(dealer, casino)\}$ cannot be successive in any order.

Selector The pipeline will choose from several courses of action to attain one of several goals: canceling a game may turn off future patrons, refusing a bet incurs the loss of future revenue, and reducing the monitor's freedom to reimburse players when the dealer defaults may further irritate some players.

Transparency Requirement	TK-LTL Formula
Maximize gains to the casino	$\widehat{C}^{\top}_{Pay(casino,\cdot)} - \widehat{C}^{\top}_{Pay(\cdot,casino)}$
Maximize the total number of bets that are placed	$\widehat{C}^{\top}_{Bet(\cdot)}$
Highest number of games run simultaneously	$\widehat{C}^{\top}_{NewGame(\cdot)} - \widehat{C}^{\top}_{EndGame(\cdot)}$
Minimize the number of bets that are placed while no games are running	$\widehat{C}^{\top}_{\mathcal{P}_{=0}\widehat{C}^{\top}_{NewGame(\cdot)}-\widehat{C}^{\top}_{EndGame(\cdot)} \wedge Bet(\cdot)}$

Table 4.1 : Representation of four possible transparency constraints using TK-LTL.

As in the museum use case, the enforcement preorder and most of the possible requirements can be formulated using simple formulas in TK-LTL. The sub-formula $\widehat{C}^{\top}_{Bet(\cdot)}$ counts the total number of bets that are placed, and can be used as a transparency constraint if the casino's main concern is to maximize the total number of bets that are placed. A monitor that seeks to achieve this goal will thus avoid suppressing bet events from the input stream. Conversely, the formula $\widehat{C}^{\top}_{Pay(casino,\cdot)} - \widehat{C}^{\top}_{Pay(\cdot,casino)}$ expresses an alternative transparency requirement, namely maximizing gains for the casino.

Two other properties could be applied for this use case. The first is maximizing the number of games run simultaneously, which can be expressed by the formula $\widehat{C}^{\top}_{NewGame(\cdot)} - \widehat{C}^{\top}_{EndGame(\cdot)}$; the second is minimizing the number of bets that are placed while no games are running and can be expressed as:

$$\widehat{C}^{\top}_{\mathcal{P}_{=0}\widehat{C}^{\top}_{NewGame(\cdot)}-\widehat{C}^{\top}_{EndGame(\cdot)} \wedge Bet(\cdot)}$$

Other goals could be possible, such as maximizing the number of different players that participate in a game, or minimizing the number of games with no or few bettors.

Table 4.1 summarizes the various enforcement preorders that could be used; and introduces two other possibilities. Each of these formulae will drive the monitor into a consequent course of action— such as premature ending games in the first case, or refusing bets in the second one, and the output of the monitor will be the optimal sequence that can be generated given the limitations of the monitor. It is important to stress that in each case, the optimal course of action is automatically selected by the monitor. It is not necessary to specifically code it, nor are elaborate proofs of correction needed.

4.4 CONCLUSION OF THE CHAPTER

In this chapter, we presented a flexible runtime enforcement framework to provide a valid replacement for any misbehaving system and guarantee that the new sequence is the optimal one with respect to an objective criterion we call *transparency constraints*. A proxy interposed between the input sequence and the monitor is used to generate all the possible replacements. A monitor then eliminates invalid options, while a selector identifies the optimal replacement sequence with respect to a transparency constraint, separate from the security policy. We described a novel formalism to state this constraint; the implementation of these concepts as an extension leveraging the BeepBeep event stream processing engine, and run through a range of different scenarios will be described in the Chapter 5.

CHAPTER V
IMPLEMENTATION AND EVALUATION

In Chapter 3, we described the runtime verification framework composed of the access proxy and the multi-monitor as well as the monitoring algorithm used by this framework to produce a multi-verdict. In Chapter 4, we endeavoured to describe the runtime enforcement model framework in an abstract way that is not tied to any specific system or formalism and to give users the freedom of choosing the formal notation of their choice for each component of the pipeline. Nevertheless, a software implementation of each framework has been developed as a Java library that extends the BeepBeep event stream processing engine [93] and several experiments are done to test each framework. This chapter presents an overview of BeepBeep, the experiments conducted, and a detailed discussion of the obtained results.

5.1 OVERVIEW OF BEEPBEEP

BeepBeep [86, 93] is a tool that can perform various tasks over event streams of different natures. The fundamental building block of BeepBeep is called *a processor*. A processor takes one or more event streams as its input, performs a computation on the elements of these streams and returns one or more event streams as its output. Several commonly used functionalities are already present across a number of palettes represented as libraries (i.e. JAR files), and the user can define new processors or functions to be used with BeepBeep's core elements.

BeepBeep has a few features that distinguish it from other event processing systems, such as being *intuitive* in the sense that any computation done in a processor can be expressed in a graphical way using a set of pictograms as in Figure 4.7. A processor object is represented by a square box, with a pictogram that indicates the type of computation it executes on events.

On the sides of this box are one or more "pipes" representing its inputs and outputs.

A third feature is having a *modular* architecture in which all of its functionalities are packed into palettes, which the user can include in their project only if they need its contents. This is in contrast to many other systems that seek to deliver a massive, one-size-fits-all set of functionalities. Customized computations are possible over event traces by allowing processors to be composed; this means that the output of a processor can be redirected to the input of another, creating complex processor chains. Events can either be *pushed* through the inputs of a chain, or *pulled* from the outputs, and BeepBeep takes care of managing implicit input and output event queues for each processor. In addition, users also have the freedom of creating their own custom processors and functions by extending the `Processor` and `Function` objects, respectively.

Extensions of BeepBeep with predefined custom objects are represented in palettes; there exist palettes for various purposes, such as signal processing, XML manipulation, plotting, and finite-state machines. BeepBeep has been used in a variety of case studies [44, 91, 114, 160, 182].

5.2 EXPERIMENTS ON MULTI-TRACES

The concepts stated in Chapter 3 have been concretely implemented as an extension to BeepBeep. Experiments with a number of different scenarios show that the multi-monitor adds constant memory overhead and linear time overhead over an input trace, which means that it can scale to large traces and large monitors (10^6 events and more than 10^9 states). Furthermore, we show that some types of data degradation can only be accounted for in related works by an over-approximation of uncertainty, which has a significant negative impact on the precision of a monitor's verdict and its performance, compared to the finer modeling presented in our work.

Finally, this model opens the door to numerous exciting theoretical questions, which we briefly enumerate in Section 3.3 as future work.

An implementation of propositional machines has been realized in the form of a Java library that extends the BeepBeep event stream processing engine [93]. The library is open source and publicly available[3]. In this library, multi-events exist in two flavors: the `ConcreteMultiEvent` is implemented as a set of valuations, while the `SymbolicMultiEvent` is implemented as a propositional formula. Both classes implement the same methods to enumerate their valuations and determine if two events have a non-empty intersection. Hence, a trace of events and a propositional machine can use either of these two multi-event types interchangeably. Propositional Mealy machines are implemented as an object that descends from BeepBeep's `Processor` class; this means that once they are instantiated, they can be connected to any other BeepBeep processor to form a potentially complex pipeline. Similarly, propositional formulas are descendants of BeepBeep's `Function` class. A downloadable instance containing all the experiments can be obtained online[4]. All the experiments were run on a Intel CORE i5-7200U 2.5 GHz running Ubuntu 18.04, inside a Java 8 virtual machine with 1746 MB of memory.

5.2.1 OVERHEAD EXPERIMENTS

A first set of experiments is meant to assess the overhead, both in terms of running time and memory consumption, incurred by the presence of an access proxy and the lifting of a uni-monitor into a multi-monitor. Our experiments are made of a number of "scenarios", where each scenario corresponds to a source of uni-events, an access proxy and a property to monitor, the latter two expressed as propositional Mealy machines:

[3]https://github.com/liflab/propositional-machines

[4]https://github.com/liflab/propositional-machines-lab

Simple: the running example represented in Figure 3.1.

MPlayer: a generated sequence of operations (play, pause, etc.) of the operation of a media player (cf. [105]), with an access proxy applying the load shedding strategy discussed in Section 2.2.2. The monitor verifies the correct ordering of the operations; it has 5 states and 20 transitions.

Temperature Threshold: a scenario made of CPU temperature readings from a cyber-physical system, adapted from [5]. Temperatures are encoded using 20 Boolean variables representing intervals of 1 degree. The access proxy applies a transformation that adds an uncertainty of ±2 degree. The monitor checks that for the first 100 units of time, whenever the temperature falls below a certain threshold T, it will again be above the threshold within 5 units of time. This monitor has 486 states and 966 transitions.

Shopping Cart: a scenario made of Boolean-encoded sequences of shopping cart manipulation operations. The monitor verifies properties on the sequence of operations based on a study of an Amazon web service [90]; it has 1,154 states and 7,267 transitions. The access proxy replaces 5% of events by the multi-event Ω symbolizing data loss.

CPU Load: a scenario made of Boolean-encoded CPU load values. The monitor uses the same property as in [125], which checks that the average load over a sliding window of five readings does not exceed some arbitrary threshold T. The access proxy adds an uncertainty of ±1% to each reading. Due to the presence of a sliding window and the use of arithmetic, this last example has a very large state space, consisting of 2×10^9 states and more than 10^{10} transitions.

For each of these scenarios, we ran a randomly-generated input trace of 100000 uni-events into the uni-monitor alone, and then into the access-controlled monitor made of the access

Table 5.1 : Global impact of the presence of an access proxy for the tested scenarios.

(a) Throughput (Hz)

Scenario	With	Without
MPlayer	42158.516	1428571.4
Shopping Cart	33277.87	609756.1
Simple	452488.7	1086956.5
Tempreature Threshold	53966.54	431034.47

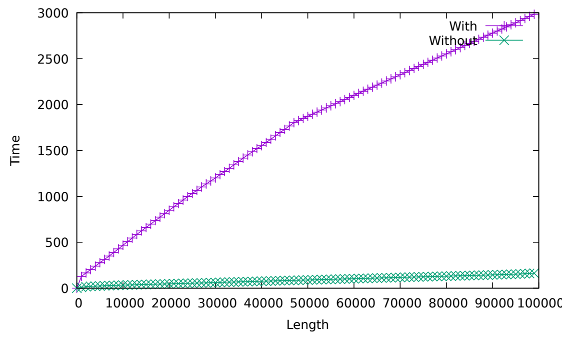
(b) Memory (B)

Scenario	With	Without
MPlayer	23622	9214
Shopping Cart	124344	119696
Simple	10722	7714
Tempreature Threshold	87600	85508

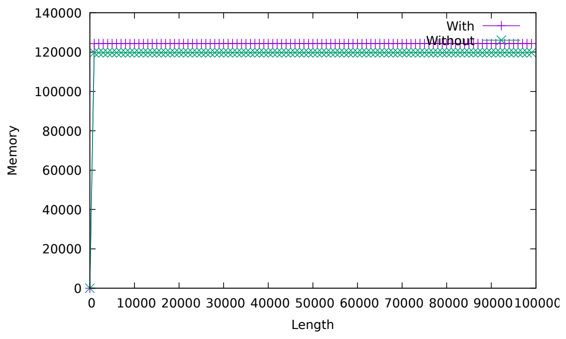
proxy and the multi-monitor. We measured the difference in terms of throughput (number of events ingested per unit of time) and memory consumption. The global impact of the presence of the access proxy is summarized in Table 5.1. In terms of throughput, it can be observed that the inclusion of an access proxy induces a slowdown on the monitoring process, since the monitor must handle multi-events instead of uni-events, and track the various possible states the uni-monitor can be in. However, for the traces and properties included in our tests, this slowdown ranges between 2× and 8×, which seems to indicate that the handling of multi-events does not impose too big an overhead on the performance of the monitor.

This should be put in perspective with the extremely large number of uni-traces handled by the multi-monitor. In the *Simple* scenario, the access proxy generates a multi-trace that corresponds to 10^{8631} distinct uni-traces; in the *Shopping Cart* scenario, this number reaches 10^{26400}. However, the complexity of Algorithm 1 does not depend on the number of uni-traces, but rather on a much simpler metric, which is the number of multi-events produced for each

Figure 5.1 : Impact of the presence of an access proxy on the throughput and memory consumption of the monitor, for the *Shopping Cart* scenario.



(a) Running time



(b) Memory consumption

input uni-event; moreover, only a count of uni-traces needs to be maintained, and uni-events are discarded after processing. This is why our approach scales despite the large number of "possible worlds" introduced by the insertion of uncertainty by the proxy.

The impact is less noticeable in terms of memory (Table 5.1b). Even though the presence of multi-events does increase the maximum amount of memory consumed by the access-controlled monitor with respect to the single uni-monitor, this increase is relatively negligible and never exceeds a factor 1.5.

To get further details on the actual behavior of the access-controlled monitor, we also measured the evolution of time and memory consumption across a trace. Running time is shown in Figure 5.2a for the *Shopping Cart* scenario (plots for the remaining scenarios are not shown but exhibit very similar trends). An important point is that processing time per event is higher, yet constant. This feature is important for an access-controlled monitor to be usable for long-running systems.

Memory consumption is plotted in Figure 5.2b. One can observe that the memory consumption of both the uni-monitor and the access-controlled monitor is constant throughout the whole trace. This observation is not surprising, as Algorithm 1 uses data structures (the mappings β and σ) of constant or bounded size. More importantly, although each input uni-event may result in multiple uni-events being processed, these events are discarded at the end of the processing and only their count needs to be kept.

5.2.2 COMPARISON TO OVER-APPROXIMATIONS

As we mentioned earlier, our modeling of imprecision and uncertainty can account for finer-grained restrictions that can only be expressed as conservative (i.e. world-preserving) over-approximations in the state of the art. To better highlight the impact that such approximations can have on the performance of a monitor, we designed a second set of experiments that revisits three of the scenarios. In each case, we describe the operation of two access proxies: the first is the one used by our approach, and the second is an over-approximation of this proxy that is the "best effort" that can be modeled by one of the related approaches mentioned in Section 2.3.

Simple: to symbolize impedance mismatch, our access proxy replaces values of $\dot{a}$ and $\dot{b}$ by the less precise assertion $\dot{a} \vee \dot{b}$. Models that do not handle correlated uncertainty (e.g. [105, 125]) must rather resort to an over-approximation where all occurrences of $\dot{a}$ and $\dot{b}$ are replaced by possible worlds where they can be either true or false whenever one of them is true in the original event.

Temperature Threshold: our proxy is left unchanged from the original set of experiments; the over-approximation replaces all variables in the temperature interval of each event by the possible worlds where they can hold any value. As discussed in Section 3.1.3.2, this approximation is necessary in a framework where all variables must be given a single ternary Boolean value (e.g. [27]).

MPlayer: to show the impact of impedance mismatch, our proxy has the *Stop* and *Pause* events conflated into a fuzzier *Interrupted* event that stands for both of them. Similar to the *Simple* scenario, the over-approximation replaces their value into possible worlds where they can both be either true or false (as would be required in e.g. [105, 125]).

The results are summarized in Table 5.2. Table 5.2a shows that the use of a coarser-grained modeling of imprecision generally has a negative impact on throughput (with the exception of *Simple*), mostly caused by the larger number of possible worlds that must be handled by the over-approximation. More interestingly, Table 5.2b shows that, as we already hinted earlier, this over-approximation also impacts the precision of the verdict returned by the underlying monitor. For each scenario, it shows the base-10 logarithm of the number of uni-projections mapped to each verdict T(rue), F(alse) and I(conclusive), for both our access-controlled proxy (P) and the "best effort" over-approximation (B).

In all scenarios, the over-approximation cannot produce a definite verdict. For *Temperature*, about 10% of all uni-projections are mapped to the "unknown" verdict instead of the correct false verdict. In comparison, our access-controlled proxy produces a single clear false verdict. For scenarios such as *Simple*, the over-approximation fares even worse: it causes all three verdicts to be possible, whereas our proposed access controlled monitor still produces a single (false) verdict. Although it can be observed that, in the over-approximation, only 0.1% of all uni-traces are mapped to the incorrect verdict, we argue there is nevertheless a fundamental qualitative gap between a definite correct verdict and a merely *likely* one, especially in the context of safety-critical systems, where monitoring is commonly employed.

The case of the *MPlayer* scenario also deserves discussion. The correct verdict of the original uni-trace should be "?". In this case, both our proxy and the over-approximation produce an equivocal verdict. However, the over-approximation makes the false verdict many

Table 5.2 : Impact of using an over-approximation for the various scenarios.

(a) Throughput (Hz)

Scenario	Our approach	Over-approx.
MPlayer	1666.6666	666.6667
Simple	1818.1818	2857.1428
Tempreature Threshold	229.88506	175.4386

(b) Verdict precision

Scenario	TP	IP	FP	TB	IB	FB
MPlayer	0	4.85	4.85	0	4.85	11.08
Simple	0	0.0	1.94	5.83	10.25	8.28
Temperature Threshold	0	0	28.82	0	59.07	60.83

orders of magnitude more likely than the (correct) inconclusive verdict, while in our proxy, both verdicts are relatively nose-to-nose. This shows that, in some cases, an over-approximation can not only result in a clear verdict being turned into an uncertain one, it can also be such that the verdict given as the most likely is the incorrect one.

5.3 EXPERIMENTS ON ENFORCEMENT

In Chapter 4, we endeavoured to describe the runtime enforcement model in an abstract way that is not tied to any specific system or formalism and to give users the freedom of choosing the formal notation of their choice for each component of the pipeline. Nevertheless, we implement the model using a Java library that extends the BeepBeep event stream processing engine [93] and present a use case. In this section, we describe the implementation procedure and the experimental evaluation and show that our proposed framework can dynamically select an adequate enforcement actions at runtime, without the need to manually define an enforcement monitor.

5.3.1 IMPLEMENTATION

The pipeline described in Section 4.3.1 has been implemented as a stand-alone BeepBeep extension. This extension, which amounts to a little more than 2,600 lines of Java code, provides a new `Processor` class (the generic entity performing stream processing in the BeepBeep) called `Gate`. This class must be instantiated by defining four parameters. The first three are the transducers μ, π, and ρ representing the monitor, the proxy, and the ranking transducer described earlier, respectively.

In line with the formal presentation of Section 4.3, the pipeline makes no assumption about the representation of these three transducers. Any chain of BeepBeep processors is accepted, provided it has the correct input/output types for its purpose. For instance, an existing BeepBeep extension called Polyglot [87] makes it possible to specify the monitor using finite-state machines, LTL, LOLA, or Quantified Event Automaton [158], while another one can be used to define the ranking transducer through a TK-LTL expression. However, the user is free to pick from all of the available BeepBeep processors to form a custom chain for any of these components. Since every Processor instance in BeepBeep can create a stateful copy of itself at any moment, the checkpointing feature required by our proposed model is straightforward to implement.

The last parameter that must be defined is the strategy that decides how the filter and selector will buffer and release events, as discussed in Section 4.3.1.5. Concretely, this is done by specifying a method named `decide`, which is called every time a new prefix tree element is received by the selector. By default, the EM accepts an integer k and picks an output trace after k calls (with $k = 1$ corresponding to the immediate greedy choice); overriding this method produces a different behavior implementing another strategy. In the experiments described later, it was arbitrarily set to $k = 8$.

The rest of the operations are automated. Once a `Gate` is instantiated, it works as a self-contained processor which, internally, operates the pipeline described in Figure 4.7. To the end user, this processor can be used as a box receiving a sequence of events in Σ and producing another sequence of events in Σ, which automatically issues corrected sequences when a policy violation occurs. It can be freely connected to other processor instances to form potentially complex computation chains.

Figure 5.2 shows a concrete example of how such a pipeline can be instantiated. First, a processor `mu` corresponding to the monitor is created. In the code example, this monitor is taken to be a Moore machine, whose states and transitions would be defined through a series of calls to a method named `addTransition` (a single example of which is shown in the excerpt). The next instruction instantiates the processor that is to act as the proxy `pi`; in this case, a processor already provided by our extension is used, called `InsertAny`. It is a predefined proxy that can, upon any input event, insert before it a fixed number of other events. The precise way in which it is instantiated in the example makes it such that either event a or event b may be inserted before any input event.

The next instruction instantiates the processor acting as the ranking transducer `rho`. This time again, a predefined processor is used (`CountModifications`), which increments the score of an input trace by one for every event that is either added or deleted. Finally, a `Gate` processor encompassing the pipeline of Figure 4.7 is created by passing as parameters the processors defined earlier. The presence of `IntervalFilter` is the processor that implements the strategy of deciding when to output a corrected segment. In this case, it is instructed to wait for at most one input event before producing a correction.

The remaining lines show how once instantiated, this `Gate` can be used like any other BeepBeep processor. That is, the gate is connected to an event sink, and events are then pushed

```
// Define the monitor verifying the policy
Processor mu = new StateMooreMachine(1, 1);
mu.addTransition(0, new EventTransition("a", 1));
...

// Define the proxy
Processor pi = new InsertAny(1, "a", "b");

// Define the selector
Processor rho = new CountModifications();

// Instantiate the pipeline with
Gate g = new Gate(mu, pi,
  new IntervalFilter(pi, 1), rho);

// Connect the gate to a sink and push events
QueueSink s = new QueueSink();
Connector.connect(g, s);
Pushable p = g.getPushableInput();
p.push("a");
p.push("a");
```

Figure 5.2 : Code usage of the runtime enforcement pipeline.

to its input in a standard BeepBeep fashion. Assuming that the policy implemented by monitor `mu` corresponds to the condition “no two successive a must be present”, the content of the sink after pushing a twice is the trace aba. It is consequent with the fact that:

1. the original input trace violates the policy;
2. the proxy is allowed to insert b anywhere in the input;
3. the filter is instructed to wait for at most 1 input event before issuing a corrected version;
4. the trace aba is indeed a corrected trace that complies with the policy.

5.3.2 SCENARIOS

As one can see in Figure 5.2, a few lines of code suffice to create an enforcement pipeline where each parameter can be an arbitrary chain of BeepBeep processors. This makes the implementation an excellent playground to experiment with various policies and proxies. Therefore, to test the implemented approach, we performed several experiments made of a number of scenarios, where each scenario corresponds to a source of events, a property to monitor, a proxy applying specific corrective actions, a filter, and a ranking selector applying a specific enforcement preorder.

The set of experiments has been encapsulated into a LabPal testing bundle [92], which is a self-contained executable package containing all the code required to rerun them [170]. For each variation of a scenario, we ran the enforcement pipeline on a randomly generated trace of length 1,000 of the corresponding type. The experiments are meant to assess the overhead, both in terms of running time and memory consumption, incurred by the presence of the proxy and the selector. All the experiments were run on an Intel CORE i5-7200U 2.5 GHz running Ubuntu 18.04, inside a Java 8 virtual machine with the default 1964 MB of memory.

In addition to the *Museum* and *Casino* use cases described earlier, our experiments include the following.

Simple An abstract scenario where the source of events is a randomly generated sequence of atomic propositions from the alphabet $\Sigma = \{a, b, c\}$. Different proxies are considered for the purpose of the experiments: adding any event at any time, deleting any event at any time, adding/deleting only event a, or adding two events at a time. These proxies are meant to illustrate the flexibility of our framework to define possible corrective actions. Similarly, various policies are also considered: one corresponding to the LTL formula $\mathbf{G}\,(a \rightarrow (\neg b\,\mathbf{U}\,c))$, another that stipulates that events a must come in pairs and the last corresponding to the regular expression $(abc)^*$. Finally, the enforcement preorder in this scenario assigns a penalty (negative score) by counting the number of inserted and deleted events in a candidate trace. This leads the pipeline to favour solutions that make the fewest possible modifications to the input trace.

File Life Cycle The second scenario is related to the operations that can be made on a resource such as a file and is a staple of runtime verification literature [49]. A trace of events is made of interleaved operations open, close, read and write on multiple files. The policy is notable in that it is *parametric*: it splits the trace into multiple sub-traces (one for each file) and stipulates that each file follows a prescribed life cycle (read and write are allowed only between open and close, and no write can occur after a read). Specifically, each event has a parameter indicating to which file this event belongs which allows to classify the events into sub-traces. The monitor for this policy is a Moore machine embedded into a BeepBeep `Slice` processor. The scenario reuses a proxy and ranking transducer from *Simple*.

5.3.3 IMPACT ON OVERHEAD

The first important measurement is the impact of the use of the runtime enforcement pipeline on the running time and memory consumption of the system.

The results on this aspect are summarized in Table 5.3. As one can see, the number of input events processed per second ranges from hundreds to thousands. Overall, one can conclude that the overhead incurred by the use of the pipeline is reasonable. For instance, in a real-world setting such as a blockchain, the limiting factor is more likely to be the number of transactions per second supported by the infrastructure itself; as a single example, the Ethereum network handles at most a few dozen transactions per second on the main net [36]. On its side, memory overhead remains relatively low with a few kilobytes, with a maximum demand of about 120 kB for a single scenario. Upon examination of the data, we observed that this corresponds to a single peak during the whole execution, with memory consumption

Table 5.3 : Summary of throughput (in events/sec.) and maximum memory consumption (in bytes) for each scenario.

Event source	Policy	Proxy	Scoring formula	Throughput	Max memory
Casino	Casino policy	Casino proxy	Maximize bets	2380	9824
			Maximize gains	490	7976
			Minimize changes	2325	8814
Files	All files life cycle	Delete any	Minimize changes	78	9580
Museum	Museum policy	Museum proxy	Maximize children	4347	9580
			Minimize changes	480	7984
			Minimize idle guards	1694	9580
a-b-c	(abc)*	Delete any	Minimize changes	628	9580
		Insert any	Minimize changes	18	8692
	After a, no c until b	Delete any	Minimize changes	869	8236
		Insert any	Minimize changes	67	119076
		Insert any b	Minimize changes	485	10344
	Stuttering a's	Delete any	Minimize changes	952	9580
		Insert any	Minimize changes	602	9396

otherwise remaining mostly below 10 kB.

Global overhead varies based on the actual combination of policy, proxy and ranking transducer. For instance, the $(abc)^*$ policy, when used on a proxy that only has the power to insert events into the trace, results in the slowest throughput. This scenario represents an extreme case since at any moment in the trace, a single next event is valid. Since the input trace is randomly generated, the probability that an input event is not the expected one is about $2/3$, meaning that the pipeline must perform corrective action on almost every event.

The action of a proxy can also be examined in further detail. Figure 5.3a shows the cumulative number of deleted, inserted and output events produced as the input trace is being read, for a variant of the museum scenario. Although difficult to see due to the scale of the plot, the output event line increases in an irregular staircase pattern. This is caused by the fact that the gate withholds events at moments when the policy is temporarily violated. One can also observe that, for this scenario, the enforcement pipeline inserts and deletes events in a relatively equal (and small) proportion.

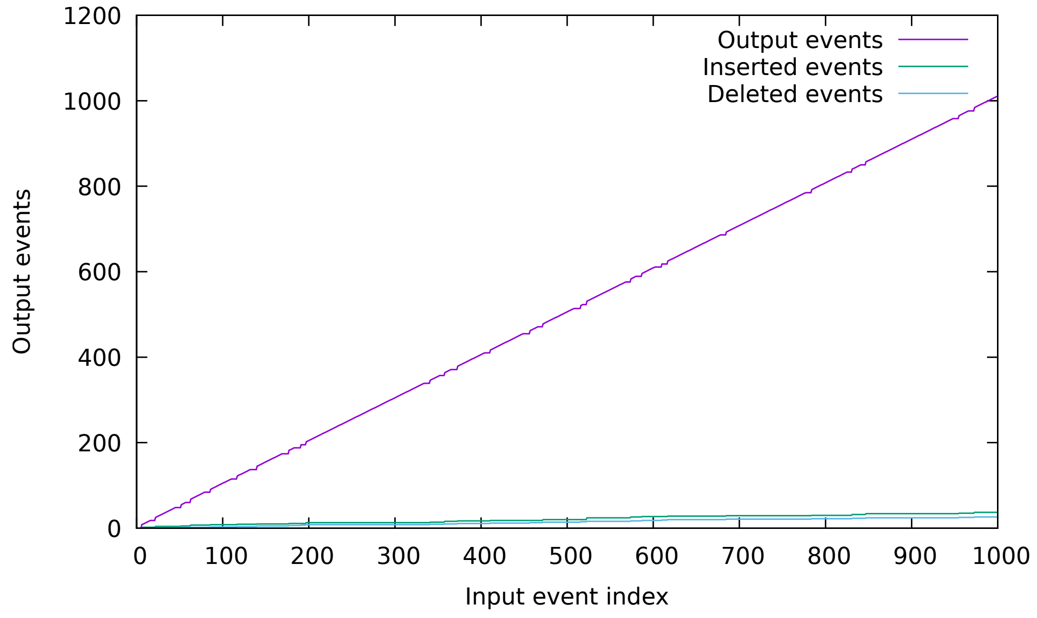
On its side, Figure 5.3b shows the memory used by the pipeline at each point in the execution. Memory remains near zero as long as the input trace does not violate the property; as a matter of fact, these flat regions exactly match the locations in Figure 5.3a where no change occurs on both inserted and deleted events. The memory plot also shows spikes, which correspond to the moments in the trace where the enforcement pipeline kicks in and starts generating possible corrected sequences. Once one such sequence is chosen and emitted, all data structures are cleared, and memory usage drops back to zero. These observations are consistent with the expected operation of the pipeline described in Section 4.3.

5.3.4 PROXY COMPARISON

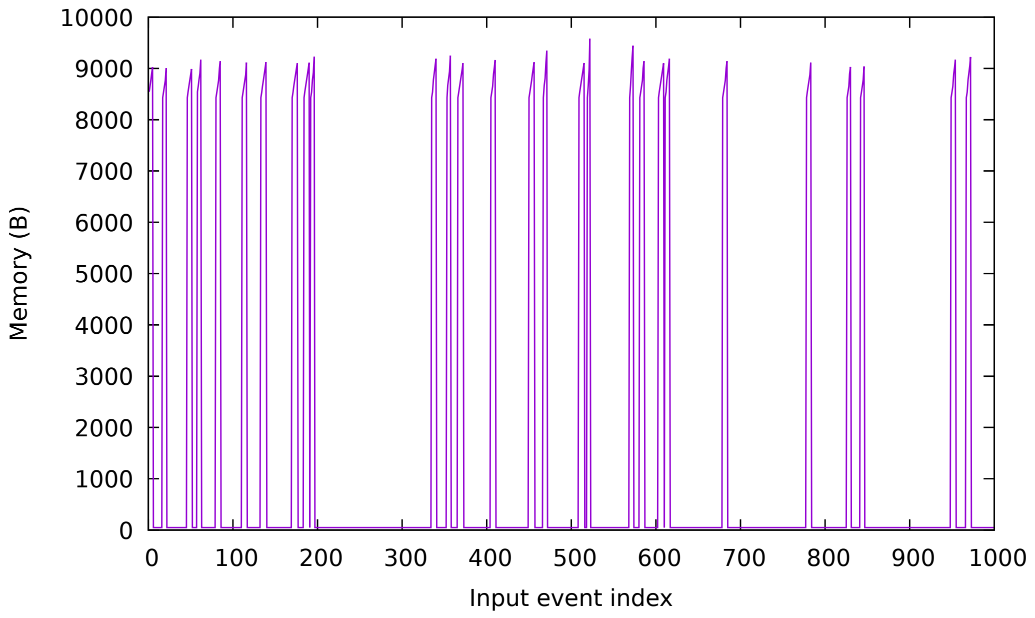
An interesting side effect of the proposed implementation is that it makes it relatively easy to compare the effect of various enforcement strategies and scoring functions for the same policy and the same input sequence. To this end, it suffices to create a different instance of the `Gate` processor and vary some of its input parameters. This section discusses such a comparison by focusing on the *Museum* scenario described earlier.

We consider four different enforcement strategies:

1. *Children shadow*: in this case, the proxy is allowed to insert a guard before every child enters the museum. It consequently takes a guard out of the museum every time a child gets out (that is, each guard "shadows" a child). Any other guard can come in but is



(a) Input vs. output events



(b) Memory consumption

Figure 5.3 : Runtime statistics for the execution of an enforcement pipeline on a variation of the museum scenario.

prevented from going out. Other events are left unmodified. This proxy is notable for being memory-less: it is not required to keep any information from the past to perform its actions.

2. *Delete children*: the proxy keeps an exact count of children and guards. It deletes any c^+ event when no guard is in the museum, and inserts as many c^- events as there are children in the museum when the last guard gets out. In other words, this proxy prevents entering or throws children out, depending on the presence of guards.

3. *Insert guard*: as with the previous proxy, this one counts children and guards. It inserts a g^+ whenever a child enters a museum with no guard inside, or when the last guard gets out and children are still in the museum. Otherwise, the input events are let through without modification.

4. *Museum proxy*: this is the proxy used in the experiments of the previous section. It has more freedom than the previous ones: if a child enters, the proxy may first insert a "guard in" event or delete the 'child in" event. If a guard exits, the proxy may delete the "guard out" event. Contrary to the previous ones, this proxy makes these modifications to the trace without any regard for the state of the policy. As per the definition of our enforcement pipeline, it is up to the downstream selector to weed out corrections made by this proxy that are not compliant with the policy. This means this proxy could not be used as a classical EM and must be encased in our proposed enforcement pipeline.

These proxies are tested against the same input trace, and their respective impact on the input trace is empirically measured by looking at the number of modifications each incurs on that input. The results are presented in Figure 5.4a. As explained in Section 4.2.2, the ranking function in such a case starts from 0 and subtracts 1 for every added or deleted event in the selected output trace. Thus, traces are assigned a negative score, with a higher value indicating fewer modifications.

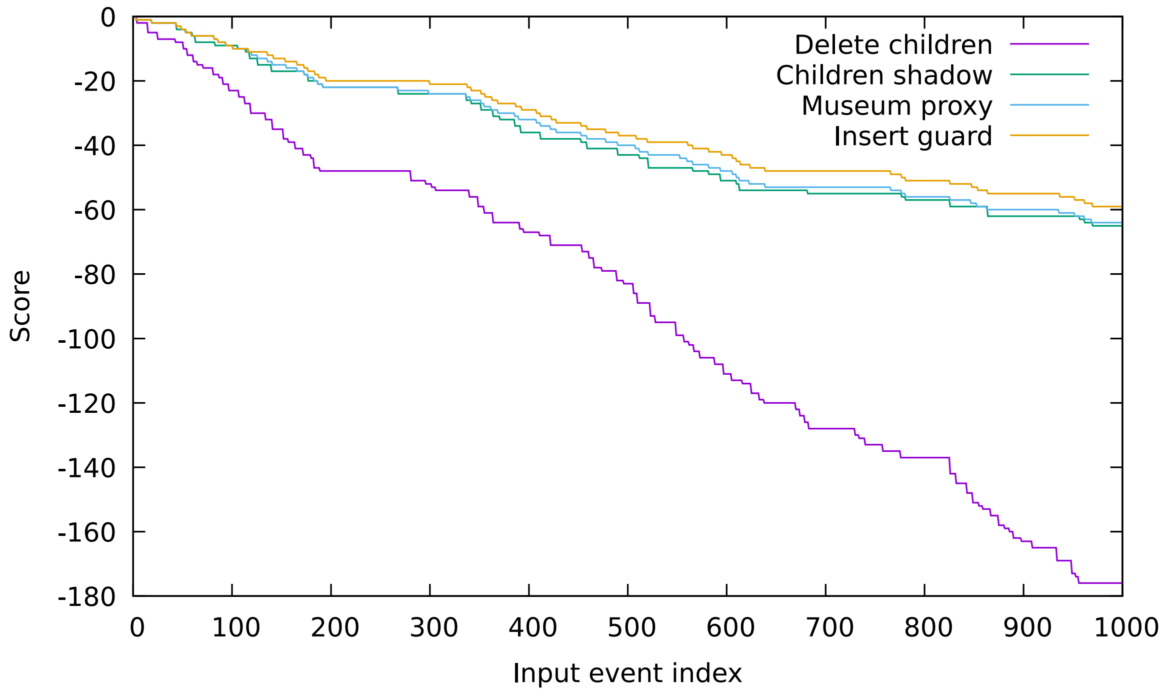
As one can see in the figure, the effect of each proxy on the trace results in different scores depending on the enforcement strategy. The *Delete children* strategy, in particular, introduces substantially more changes to the input trace than the remaining ones. This is expected, as when the last guard comes out, all children currently in the museum are expelled at once, resulting in a potentially large number of c^- being inserted into the corrected trace. In contrast, other proxies exhibit a less invasive (and ultimately roughly equivalent) behavior on the input trace.

Note, however, that this impact depends on the proxy but also on the enforcement preorder. This is exemplified in Figure 5.4b, which trades function ρ for a new version where each trace starts with a score of 0, and is decremented by 1 for each time step where guards are in the museum without any children. Note how this preorder is uncorrelated to the number of changes being made to the input: a large number of modifications will be deemed preferable if it ensures a smaller number of idle guards in the museum. According to this metric, this time, the proxies are reversed. In this case, *Delete children* turns out to be the proxy producing higher-scoring corrections than the others.

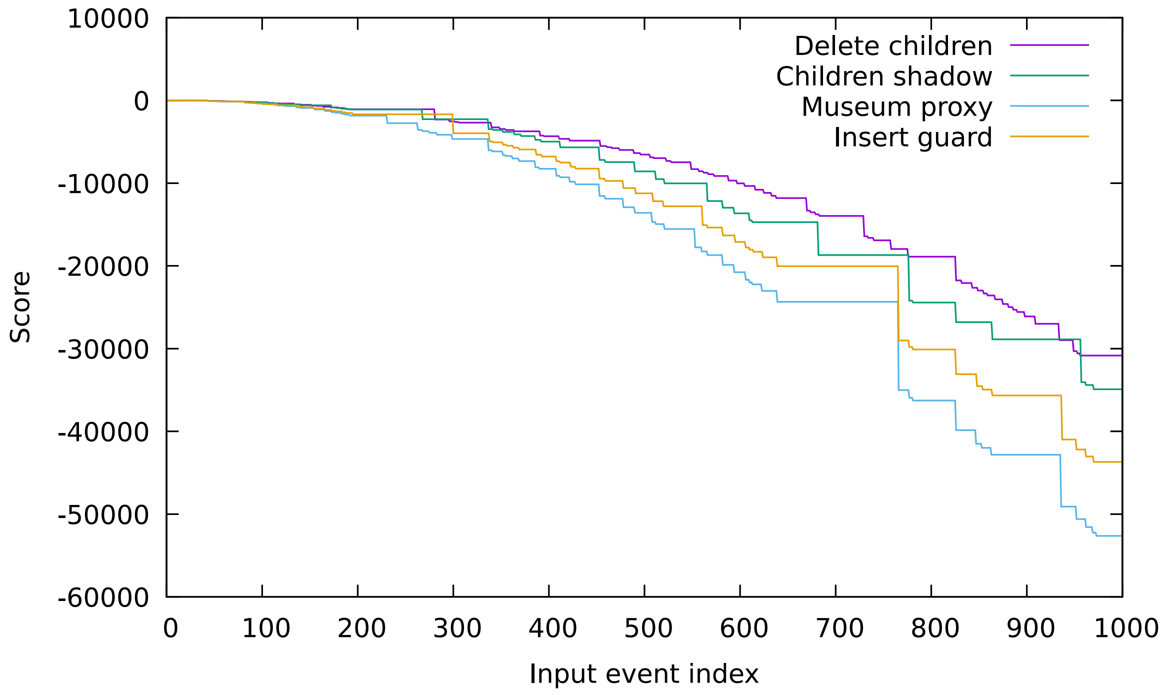
5.3.5 RESULTS AND ANALYSIS

This proposed model can be seen as a generalization of the multi-trace model presented in Section 3.1, which handles uncertainty and missing events as sets of possible worlds called "multi-events". The difference between the two models lies in the fact that a multi-event contains multiple single events, while a sequence set contains multiple event sequences.

The pipeline proposed in Figure 4.7 should be contrasted with the classical EMs



(a) Minimize changes



(b) Minimize idle guards

Figure 5.4 : The comparison of the action of different proxies on the same input sequence, for two ranking functions.

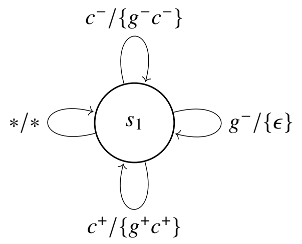


Figure 5.5 : An EM applying the *Children shadow* correction strategy in the *Museum* example.

considered in past literature, which takes the form of Figure 2.2. In an EM, an input sequence is transformed into a corrected output sequence in a single step. It is up to the EM to keep track of the specification's current state, decide on the appropriate modifications to apply to each incoming input event (including possibly buffering this event and deciding later) and produce a *single* output trace that must be guaranteed to satisfy the policy. As we have seen, automatic synthesis algorithms for such EMs are rare, entailing that they must typically be designed by hand for each policy and set of available corrective actions. Alas, this task turns out to be nontrivial even for simple cases, and formally proving that an EM always produces a valid output regardless of its input is equally challenging.

In addition, our definition of a correction, introduced in Section 4.2.1, is different from what is typically expected of an EM. To illustrate this, consider the EM illustrated in Figure 5.5. Given the input trace $a^+a^+c^+c^+$, which violates the museum policy (no child in without a guard), it produces the output trace $a^+a^+g^+c^+g^+c^+$. One can check that this sequence indeed satisfies the policy, and moreover that the prefix of the input that satisfies the policy (a^+a^+) has been output without modification (as required by the basic transparency expectation).

However, this EM does *not* produce an output sequence that satisfies our definition of a correction. After ingesting the first three events, the EM produces the sequence $a^+a^+g^+c^+$, inserting the g^+ event required to restore satisfaction of the policy. But then, since this corrected output places the policy back in a valid state, the next input event (c^+) does not introduce a violation. Following the terminology of Section 4.2.1, it represents a segment that is positive throughout and thus must be output without modification. Thus, to abide by the definition of a correction, one cannot simply add a guard before every child.[5] This simple example illustrates that our proposed definition of a corrected trace is tighter than existing requirements on an EM, and narrows the amount of intervention that one is allowed to make on the input.

It is also important to stress that in our proposed pipeline, the proxy only models the enforcement capabilities of a monitor, irrespective of the actual property that is meant to be enforced. That is, if an EM is allowed to remove any event from the trace, then the proxy will generate output traces where each event may or may not be present. Stated differently, the goal of the proxy is to generate all the possible modifications of the input trace that are potentially available to enforce a given property.

This generic definition presents a few advantages. First, it is agnostic to the actual representation of the enforcement capabilities. Figure 4.3 shows an example of a proxy that applies a suppression modification action; given the input trace $\sigma = babbc$, it produces the output $\{b\}, \{a\}, \{b\}, \{\epsilon, b\}, \{c\}$. An interesting feature of this model is to enable "non-standard" enforcement capabilities. For instance, a classical delete automaton can delete any event at any moment. Our abstract definition of a proxy could express a finer-grained capability, such as the fact that only successive b events following an initial b may be deleted (illustrated by the Mealy machine of Figure 4.3). Since the proxy is not tied to a specific notation and has the leeway to output any sequence set it wishes, it offers a high capacity to precisely circumscribe available enforcement actions.

[5]When used as a proxy in our enforcement pipeline, the Mealy machine of Figure 5.5 does not create this issue, as on the second c^+ input event, control is not switched to the enforcement pipeline as no violation is detected.

The modular design of the enforcement pipeline offers several advantages. Notably, it simplifies the creation of the monitor, since the process of manipulating the sequence is now separate from the process of selecting a valid replacement. The main benefit of the method we propose is that the behavior of the EM need not be coded explicitly. Instead, the behavior of the EM is simply the result of the selector seeking to optimize the evaluation of the enforcement preorder.

The model also makes it possible to select the optimal replacement sequence, according to a criterion separate from the security policy, which can be stated in a distinct formalism. The model also allows users to compare multiple alternative corrective enforcement actions and select the optimal one with respect to an objective gradation. Finally, since the alteration of the input trace is done independently of its downstream verification for compliance with the policy, the model also does away with the need for proof of the correctness of the synthesized EM, as is usually done in related works on the subject.

As we also stressed in Section 4.3, the proposed architecture is independent of the formal representation of each component. In fact, we deliberately chose three different notations for the proxy, the monitor and the ranking transducer of the casino use case to illustrate this feature. As with previous phases, the model leaves open the question of how ρ is specified. In principle, any formal model could be used to state the transparency requirement. For the enforcement preorder, several formalisms could potentially be used, including Lola [60], fuzzy-time LTL [82], or TK-LTL [111].

Some examples, taken from the literature, illustrate the flexibility of the approach. Recall the "no send after read" policy introduced in Section 2.5.1. As discussed above, the policy can be expressed by inserting an entry in the log, suppressing the send event, suppressing the read event or by aborting the execution. In this case, the property would be enforced by assigning a value to each trace, based on present behavior (a truncated trace being naturally less valuable than a longer trace). This flexibility makes it possible to support other types of enforcement requirements. For instance, consider a monitor whose objective is to produce a valid output that is as close to the input as possible. This is a fairly intuitive requirement but difficult to implement using existing solutions. In the proposed framework, this requirement can be enforced by assigning a cost to each transformation performed by the monitor (adding an event or suppressing an event) and having the monitor minimize the overall enforcement cost for the entire sequence. Furthermore, flexibility can be achieved by assigning a different cost to each action as needed or by assigning a different cost to suppression and insertion.

CHAPTER VI
CONCLUSION

In this thesis, we have presented a flexible framework for dealing with access restrictions on events in a trace. We utilize a stateful proxy to model known gaps, imprecise values, and other forms of uncertainty in the events before they are passed to the monitor. Additionally, we have introduced a construction of a loss-tolerant multi-monitor from a uni-monitor, which can process a multi-trace and produce a multi-verdict. The likelihood of each possible verdict is quantified. Experimental results from various scenarios demonstrate that the multi-monitor incurs constant memory overhead and linear time overhead over an input trace, enabling scalability to large traces and monitors (on the order of 10^6 events and more than 10^9 states). Furthermore, we have shown that certain types of data degradation can only be addressed in related works through an over-approximation of uncertainty, which significantly impacts the precision and performance of a monitor compared to the finer modeling presented in this thesis.

Our proposed framework opens up avenues for further research questions centered around the concept of ambiguity. One such question is determining the *decidability* of ambiguity: given an access proxy π_A and a monitor π_P, can we determine if there exists a trace for which $\mathcal{M}(\pi_A, \pi_P)$ is ambiguous? This question can be linked to existing results on monitorability. Another question is related to *resolving* ambiguity: finding the minimal modifications required to π_A in order to eliminate ambiguity for a given monitor. Lastly, the reverse question of *introducing* ambiguity could be explored: given a monitor π_P, finding the "least disruptive" proxy π_A such that $\mathcal{M}(\pi_A, \pi_P)$ becomes ambiguous. This investigation could help determine what access restrictions should be introduced to prevent an attacker from deducing a sensitive property π_P from a log.

While the presented access restrictions have all been world-preserving, our model of an access proxy can also be used to apply transformations to an input trace that do not satisfy this condition. This highlights an interesting side effect of having an explicit model of event degradation, as it allows us to study the impact of feeding a monitor with a trace that is not only imprecise but also *incorrect* according to a systematic pattern. For example, we could define a proxy that removes events based on a specific pattern to examine the impact of undetected dropped events. Similarly, we could introduce a proxy that adds events based on a pattern (e.g., to study the impact of introducing stuttering into the trace). In such cases, a multi-monitor receiving such traces can no longer guarantee the soundness of its multi-verdict. Restoring soundness of the multi-monitor without relying on the world-preservation assumption is a topic for future work. One possible refinement in this scenario could involve the concept of throttling mentioned in Section 2.2.2. A monitor could be given an access "budget" for the events in a log, where each access request consumes a portion of this budget. Depending on the property and its current state, a monitor could decide whether or not to request access for an event, optimizing the use of the access budget. This would turn monitoring into an optimization problem.

A direct extension of our access control model would be the symbolic manipulation of infinite or continuous variables. This would allow for a more convenient expression of a wider range of event types and access restrictions. Our representation of events as simple boolean variables allows an alphabet of 1,000 different events to be encoded using only 10 Boolean variables. However, in cybersecurity, an event has multiple parameters, each of which has integer values, so this is hardly applicable to real life problems. A future work should incorporate a more efficient representation of multi-events. Additionally, the notion of uncertainty and loss tolerance could be extended to other formal notations beyond Mealy machines, such as Linear Temporal Logic.

Our thesis goes beyond the simple verification of a property over an incomplete trace and the generation of multiple outcomes. Instead, we present a flexible framework for enforcing a policy on a trace, ensuring that the resulting sequence of events adheres to predefined "transparency constraints." To achieve this, we employ a proxy positioned between the input sequence and the monitor, generating all possible replacements. The monitor then filters out invalid options, while a selector identifies the best replacement sequence based on the transparency constraint, which is distinct from the security policy. By applying this framework to various scenarios, we demonstrate that property enforcement can be dynamically performed at runtime without the need for manually defining a monitoring logic tailored to each specific use case.

The precise behavior of the pipeline emerges from the interplay of its components. Furthermore, we emphasize how this modular design enables easy replacement of any element of the framework (policy, proxy, preorder) with another. In fact, each individual transducer used in the scenarios benchmarked in Section 5.3 requires only a few dozen lines of code at most. This generality paves the way for future studies on a wide range of enforcement mechanisms under a unified formal framework, allowing for a more detailed comparison of their respective advantages. It should also be noted that, for many of the experimentally tested scenarios, most of the considered proxies are granted significant flexibility in modifying the trace, such as inserting or deleting events at any moment. This naturally incurs runtime overhead due to the generation of a large number of potential corrected traces. Thus, exploring proxies with tighter enforcement capabilities is a possible avenue to consider.

An important contribution of this thesis is the utilization of several categories of proxies from the literature, employing various enforcement strategies and scoring functions. We empirically compare the effectiveness of these proxies when used with the same policy and input sequence. Additionally, the thesis introduces the notion of trace correction, comparing the enforcement pipeline to enforcement using an automata model that strictly adheres to the policy. We demonstrate how this notion limits the degree of intervention a proxy can make on an input trace compared to an EM. Furthermore, it is important to note that existing enforcement mechanisms typically propose a single correction to a trace, making them degenerate forms of combinatorial proxies. In contrast, our proxy can suggest multiple corrections to the same input trace. Therefore, our combinatorial proxy generalizes any other approach proposed in the literature so far.

Furthermore, our model can be extended and enhanced in various ways. For example, it can be expanded to evaluate multiple transparency requirements over traces. The pipeline shown in Figure 4.7 can be modified by incorporating multiple ranking transducers, where each transducer assesses a specific transparency requirement and assigns a numerical score to each output trace from the proxy based on the enforcement preorder. Additionally, there is potential to relax the classical definition of transparency and allow modifications to a trace that are not solely triggered by hard policy violations.

REFERENCES

[1] "Rule-based access control," in *Encyclopedia of Cryptography and Security, 2nd Ed*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, p. 1072.

[2] "Baum-Welch algorithm," in *Encyclopedia of Machine Learning and Data Mining*, C. Sammut and G. I. Webb, Eds. Springer, 2017, p. 99.

[3] "1849-2016 - IEEE standard for extensible event stream (XES) for achieving interoperability in event logs and event streams," 2016.

[4] L. Aceto, A. Achilleos, A. Francalanza, and A. Ingólfsdóttir, "Monitoring for silent actions," in *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India*, 2017, pp. 7:1–7:14.

[5] A. Aerts, M. Reniers, and M. Mousavi, "Chapter 19 - model-based testing of cyber-physical systems," in *Cyber-Physical Systems*, ser. Intelligent Data-Centric Systems, H. Song, D. B. Rawat, S. Jeschke, and C. Brecher, Eds. Boston: Academic Press, 2017, pp. 287–304. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128038017000195

[6] T. Agarwal. (2020) Finite state machine: Mealy state machine and moore state machine.

[7] M. Aghaei, L. Baresi, and C. Ghezzi, "Medusa: a runtime verification framework for data-centric applications," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 89–99.

[8] G. Ahn, "Discretionary access control," in *Encyclopedia of Database Systems, Second Edition*, L. Liu and M. T. Özsu, Eds. Springer, 2018.

[9] N. Alechina, M. Dastani, and B. Logan, "Norm approximation for imperfect monitors," in *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, May 5-9, 2014*, A. L. C. Bazzan, M. N. Huhns, A. Lomuscio, and P. Scerri, Eds. IFAAMAS/ACM, 2014, pp. 117–124.

[10] V. Alturi and D. F. Ferraiolo, "Role-based access control," in *Encyclopedia of Cryptography and Security, 2nd Ed*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, pp. 1053–1055.

[11] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[12] Amazon Services, "Throttling: Limits to how often you can submit requests," https://docs.developer.amazonservices.com/en_US/dev_guide/DG_Throttling.html, Accessed March 14th, 2020.

[13] D. Ancona, F. Dagnino, and L. Franceschini, "A formalism for specification of java API interfaces," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, J. Dolby, W. G. J. Halfond, and A. Mishra, Eds. ACM, 2018, pp. 24–26. [Online]. Available: https://doi.org/10.1145/3236454.3236476

[14] M. Arnold, M. T. Vechev, and E. Yahav, "QVM: an efficient runtime for detecting defects in deployed systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 1, pp. 2:1–2:35, 2011.

[15] V. Arora, F. van Breugel, P. Fischer, and A. Gorbenko, "Monitoring CSV data using multi-parametric run-time interval logic," in *IEEE International Conference on Software Engineering and Formal Methods, SEFM 2017, Trento, Italy, September 6-10, 2017*, 2017, pp. 283–298.

[16] A. Artikis, T. Eiter, A. Margara, and S. Vansummeren, "Foundations of Composite Event Recognition (Dagstuhl Seminar 20071)," *Dagstuhl Reports*, vol. 10, no. 2, pp. 19–49, 2020. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/13058

[17] D. P. Attard and A. Francalanza, "A monitoring tool for a branching-time logic," in *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, ser. Lecture Notes in Computer Science, Y. Falcone and C. Sánchez, Eds., vol. 10012. Springer, 2016, pp. 473–481.

[18] M. Avalle, A. Pironti, and R. Sisto, "Formal verification of security protocol implementations: A survey," *Form. Asp. Comput.*, vol. 26, no. 1, p. 99–123, jan 2014.

[Online]. Available: https://doi.org/10.1007/s00165-012-0269-9

[19] T. Babiak, F. Blahoudek, J. Křet'ınsk'y, and D. Štill, "ltl2dstar: A tool for ltl synthesis," in *Proceedings of the 24th International Conference on Computer Aided Verification*. Springer, 2012, pp. 571–577.

[20] G. Bacci, M. Bartoletti, A. M. Moggi, and E. Tuosto, "Axml: A tool for runtime verification of xml documents," in *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*. Springer, 2011, pp. 228–232.

[21] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008.

[22] L. Baresi, M. Cominetti, and M. Rossi, "Jrec: A framework for runtime monitoring of web services," in *Fourth International Conference on Service Oriented Computing (ICSOC'06)*. IEEE, 2006, pp. 479–488.

[23] D. Barrera, D. Perez-Palacin, R. Barrado, J. Calvo-Manzano, T. San Feliu, and J. Garcia-Garcia, "Flint: Fast log inspection for runtime verification of complex system interactions," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 447–457.

[24] E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok, and J. Seyster, "Adaptive runtime verification," in *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Qadeer and S. Tasiran, Eds., vol. 7687. Springer, 2012, pp. 168–182.

[25] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to runtime verification," in *Lectures on Runtime Verification - Introductory and Advanced Topics*, ser. Lecture Notes in Computer Science, E. Bartocci and Y. Falcone, Eds. Springer, 2018, vol. 10457, pp. 1–33.

[26] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu, "Monitoring metric first-order temporal properties," vol. 62, no. 2, 2015.

[27] D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu, “Monitoring compliance policies over incomplete and disagreeing logs,” in *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Qadeer and S. Tasiran, Eds., vol. 7687. Springer, 2012, pp. 151–167.

[28] ——, “On real-time monitoring with imprecise timestamps,” in *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, ser. Lecture Notes in Computer Science, B. Bonakdarpour and S. A. Smolka, Eds., vol. 8734. Springer, 2014, pp. 193–198.

[29] D. A. Basin, F. Klaedtke, and E. Zalinescu, “Runtime verification of temporal properties over out-of-order data streams,” in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Majumdar and V. Kuncak, Eds., vol. 10426. Springer, 2017, pp. 356–376.

[30] A. Bauer and J. Jürjens, “Runtime verification of cryptographic protocols,” *Computers & Security*, vol. 29, no. 3, pp. 315–330, 2010, special issue on software engineering for secure systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404809001047

[31] A. Bauer, M. Leucker, and C. Schallhart, “The good, the bad, and the ugly, but how ugly is ugly?” in *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, ser. Lecture Notes in Computer Science, O. Sokolsky and S. Tasiran, Eds., vol. 4839. Springer, 2007, pp. 126–138.

[32] ——, “Runtime verification for LTL and TLTL,” *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 14:1–14:64, 2011. [Online]. Available: https://doi.org/10.1145/2000799.2000800

[33] L. Bauer, J. Ligatti, and D. Walker, “More enforceable security policies,” in *In Foundations of Computer Security*, 2002.

[34] D. Beauquier, J. Cohen, and R. Lanotte, “Security policies enforcement using finite and pushdown edit automata,” *Int. J. Inf. Sec.*, vol. 12, no. 4, pp. 319–336, 2013.

[35] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[36] Q. Betti, B. Montreuil, R. Khoury, and S. Hallé, *Smart Contracts-Enabled Simulation for Hyperconnected Logistics*. Cham: Springer International Publishing, 2020, pp. 109–149.

[37] N. Bielova and F. Massacci, "Do you really mean what you actually enforced?" *Int. J. of Inf. Security*, pp. 1–16, 2011, 10.1007/s10207-011-0137-2. [Online]. Available: http://dx.doi.org/10.1007/s10207-011-0137-2

[38] ——, "Iterative enforcement by suppression: Towards practical enforcement theories," *Journal of Computer Security*, vol. 20, no. 1, 2012.

[39] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 911–938, 2012.

[40] E. Bodden and K. Havelund, "Racer: effective race detection using aspectj," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, B. G. Ryder and A. Zeller, Eds. ACM, 2008, pp. 155–166.

[41] E. Bodden, L. J. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with tracematches," *J. Log. Comput.*, vol. 20, no. 3, pp. 707–723, 2010. [Online]. Available: https://doi.org/10.1093/logcom/exn077

[42] B. Bonakdarpour, S. Navabpour, and S. Fischmeister, "Sampling-based runtime verification," in *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. J. Butler and W. Schulte, Eds., vol. 6664. Springer, 2011, pp. 88–102.

[43] L. Bouganim and Y. Guo, "Database encryption," in *Encyclopedia of Cryptography and Security, 2nd Ed*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, pp. 307–312.

[44] M. R. Boussaha, R. Khoury, and S. Hallé, “Monitoring of security properties using beepbeep,” in *Foundations and Practice of Security – 10th International Symposium, FPS 2017, Nancy, France, October 23-25, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, A. Imine, J. M. Fernandez, J. Marion, L. Logrippo, and J. García-Alfaro, Eds., vol. 10723. Springer, 2017, pp. 160–169. [Online]. Available: https://doi.org/10.1007/978-3-319-75650-9_11

[45] P. Cerný, T. A. Henzinger, and A. Radhakrishna, “Quantitative simulation games,” in *Time for Verification, Essays in Memory of Amir Pnueli*, ser. Lecture Notes in Computer Science, Z. Manna and D. A. Peled, Eds., vol. 6200. Springer, 2010, pp. 42–60.

[46] H. Chabot, R. Khoury, and N. Tawbi, “Extending the enforcement power of truncation monitors using static analysis,” vol. 30, no. 4, jun 2011, pp. 194–207.

[47] E. Chang, Z. Manna, and A. Pnueli, “The safety-progress classification,” in *Logic and Algebra of Specification*, F. L. Bauer, W. Brauer, and H. Schwichtenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 143–202.

[48] F. Chen and G. Rosu, “Java-MOP: A monitoring oriented programming environment for java,” in *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, ser. Lecture Notes in Computer Science, N. Halbwachs and L. D. Zuck, Eds., vol. 3440. Springer, 2005, pp. 546–550.

[49] F. Chen, P. O. Meredith, D. Jin, and G. Rosu, “Efficient formalism-independent monitoring of parametric properties,” in *ASE*. IEEE Computer Society, 2009, pp. 383–394.

[50] K. Chen and L. Liu, “Privacy preserving data classification with rotation perturbation,” in *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), 27-30 November 2005, Houston, Texas, USA*. IEEE Computer Society, 2005, pp. 589–592.

[51] ——, “Geometric data perturbation for privacy preserving outsourced data mining,” *Knowl. Inf. Syst.*, vol. 29, no. 3, pp. 657–695, 2011.

[52] I. Clark, "Role-based access control," in *Encyclopedia of Information Assurance*, R. Herold, Ed. Taylor & Francis, 2011.

[53] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking," in *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, M. Broy, Ed., 1996, pp. 305–349.

[54] C. Colombo, G. J. Pace, and G. Schneider, "LARVA—safer monitoring of real-time java programs (tool paper)," in *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23–27 November 2009*, D. V. Hung and P. Krishnan, Eds. IEEE Computer Society, 2009, pp. 33–37. [Online]. Available: https://doi.org/10.1109/SEFM.2009.13

[55] C. Colombo, M. Pradella, and M. Rossi, "Xmonitor: A runtime verification tool for xml documents," in *7th International Conference on Runtime Verification (RV 2017)*. Springer, 2017, pp. 226–233.

[56] C. Colombo, J. Ellul, and G. J. Pace, "Contracts over smart contracts: Recovering from violations dynamically," in *ISoLA 2018*, ser. LNCS, T. Margaria and B. Steffen, Eds., vol. 11247. Springer, 2018, pp. 300–315.

[57] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, "Tessla: Temporal stream-based specification language," in *Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings*, ser. Lecture Notes in Computer Science, T. Massoni and M. R. Mousavi, Eds., vol. 11254. Springer, 2018, pp. 144–162. [Online]. Available: https://doi.org/10.1007/978-3-030-03044-5_10

[58] G. Cormode and D. Srivastava, "Anonymized data: Generation, models, usage," in *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, and V. J. Tsotras, Eds. IEEE Computer Society, 2010, pp. 1211–1212.

[59] M. d'Amorim and K. Havelund, "Event-based runtime verification of java programs," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[60] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "LOLA: runtime monitoring of synchronous systems," in *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, 2005, pp. 166–174.

[61] S. D. C. di Vimercati, "Discretionary access control policies (DAC)," in *Encyclopedia of Cryptography and Security, 2nd Ed*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, pp. 356–358.

[62] S. D. C. di Vimercati and P. Samarati, "Mandatory access control policy (MAC)," in *Encyclopedia of Cryptography and Security, 2nd Ed*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, p. 758.

[63] E. Dolzhenko, J. Ligatti, and S. Reddy, "Modeling runtime enforcement with mandatory results automata," *Int. J. Inf. Sec.*, vol. 14, no. 1, pp. 47–60, 2015. [Online]. Available: https://doi.org/10.1007/s10207-014-0239-8

[64] P. Drábik, F. Martinelli, and C. Morisset, "Cost-aware runtime enforcement of security policies," in *STM*, ser. LNCS, A. Jøsang, P. Samarati, and M. Petrocchi, Eds., vol. 7783. Springer, 2012, pp. 1–16.

[65] ——, "A quantitative approach for inexact enforcement of security policies," in *ISC 2012*, ser. LNCS, D. Gollmann and F. C. Freiling, Eds., vol. 7483. Springer, 2012, pp. 306–321.

[66] D. D'Souza and P. Thiagarajan, "Synthesis of non-deterministic automata from temporal logic specifications," *Formal Methods in System Design*, vol. 17, no. 1, pp. 5–30, 2000.

[67] U. Erlingsson, "The inlined reference monitor approach to security policy enforcement," Ph.D. dissertation, USA, 2004, aAI3114521.

[68] A. Estes, "Access control matrix," in *Encyclopedia of Cryptography and Security, 2nd Ed*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, pp. 12–13.

[69] Y. Falcone, "You should better enforce than verify," in *Runtime Verification - First In-*

ternational Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings, ser. Lecture Notes in Computer Science, vol. 6418. Springer, 2010, pp. 89–105.

[70] Y. Falcone and G. Salaün, "Runtime enforcement with reordering, healing, and suppression," in *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings*, ser. Lecture Notes in Computer Science, R. Calinescu and C. S. Pasareanu, Eds., vol. 13085. Springer, 2021, pp. 47–65. [Online]. Available: https://doi.org/10.1007/978-3-030-92124-8_3

[71] Y. Falcone, J. Fernandez, and L. Mounier, "Runtime verification of safety-progress properties," in *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, ser. Lecture Notes in Computer Science, S. Bensalem and D. A. Peled, Eds., vol. 5779. Springer, 2009, pp. 40–59.

[72] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier, "Runtime enforcement monitors: Composition, synthesis, and enforcement abilities," *Form. Methods Syst. Des.*, vol. 38, no. 3, p. 223–262, Jun. 2011.

[73] Y. Falcone, J. Fernandez, and L. Mounier, "What can you verify and enforce at runtime?" *Int. J. Softw. Tools Technol. Transf.*, vol. 14, no. 3, pp. 349–382, 2012.

[74] Y. Falcone, S. Krstic, G. Reger, and D. Traytel, "A taxonomy for classifying runtime verification tools," in *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, ser. Lecture Notes in Computer Science, C. Colombo and M. Leucker, Eds., vol. 11237. Springer, 2018, pp. 241–262.

[75] Y. Falcone, L. Mariani, A. Rollet, and S. Saha, "Runtime failure prevention and reaction," in *Lectures on Runtime Verification – Introductory and Advanced Topics*, ser. LNCS, E. Bartocci and Y. Falcone, Eds. Springer, 2018, vol. 10457, pp. 103–134.

[76] L. Fei and S. P. Midkiff, "Artemis: practical runtime monitoring of applications for execution anomalies," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, M. I. Schwartzbach and T. Ball, Eds. ACM, 2006, pp. 84–95.

[77] A. Ferrando and V. Malvone, "Runtime verification with imperfect information through

indistinguishability relations," in *Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings*, ser. Lecture Notes in Computer Science, B. Schlingloff and M. Chai, Eds., vol. 13550. Springer, 2022, pp. 335–351.

[78] B. Finkbeiner and M. Schewe, "Efficient translation of ltl formulae into deterministic büchi automata," in *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation.* Springer, 2006, pp. 53–67.

[79] P. W. L. Fong, "Access control by tracking shallow execution history," in *2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA*. IEEE Computer Society, 2004, pp. 43–55.

[80] D. Fortin-Simard, J. Bilodeau, K. Bouchard, S. Gaboury, B. Bouchard, and A. Bouzouane, "Exploiting passive RFID technology for activity recognition in smart homes," *IEEE Intell. Syst.*, vol. 30, no. 4, pp. 7–15, 2015.

[81] H. Franco and A. J. Serralheiro, "A new discriminative training algorithm for hidden markov models," in *The First International Conference on Spoken Language Processing, ICSLP 1990, Kobe, Japan, November 18-22, 1990.* ISCA, 1990.

[82] A. Frigeri, L. Pasquale, and P. Spoletini, "Fuzzy time in linear temporal logic," *ACM Trans. Comput. Log.*, vol. 15, no. 4, pp. 30:1–30:22, 2014.

[83] H. Garavel and R. Mateescu, "SEQ.OPEN: A tool for efficient trace-based verification," in *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, ser. Lecture Notes in Computer Science, S. Graf and L. Mounier, Eds., vol. 2989. Springer, 2004, pp. 151–157.

[84] R. Gerth, D. A. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*, ser. IFIP Conference Proceedings, P. Dembinski and M. Sredniawa, Eds., vol. 38. Chapman & Hall, 1995, pp. 3–18.

[85] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A

vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.

[86] S. Hallé and R. Khoury, "Event stream processing with beepbeep 3," in *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*, ser. Kalpa Publications in Computing, G. Reger and K. Havelund, Eds., vol. 3. EasyChair, 2017, pp. 81–88. [Online]. Available: https://doi.org/10.29007/4cth

[87] ——, "Writing domain-specific languages for BeepBeep," in *RV*, ser. LNCS, C. Colombo and M. Leucker, Eds., vol. 11237. Springer, 2018, pp. 447–457.

[88] S. Hallé and R. Villemaire, "Runtime verification for the web - A tutorial introduction to interface contracts in web applications," in *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, ser. Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, Eds., vol. 6418. Springer, 2010, pp. 106–121.

[89] ——, "Runtime enforcement of web service message contracts with data," *IEEE Trans. Serv. Comput.*, vol. 5, no. 2, pp. 192–206, 2012.

[90] S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire, "Runtime verification of web service interface contracts," *IEEE Computer*, vol. 43, no. 3, pp. 59–66, 2010.

[91] S. Hallé, S. Gaboury, and B. Bouchard, "Activity recognition through complex event processing: First findings," in *Artificial Intelligence Applied to Assistive Technologies and Smart Environments, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016*, ser. AAAI Technical Report, B. Bouchard, S. Giroux, A. Bouzouane, and S. Gaboury, Eds., vol. WS-16-01. AAAI Press, 2016. [Online]. Available: http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12561

[92] S. Hallé, R. Khoury, and M. Awesso, "Streamlining the inclusion of computer experiments in a research paper," *Computer*, vol. 51, no. 11, pp. 78–89, 2018.

[93] S. Hallé, *Event stream processing with BeepBeep 3: Log crunching and analysis made easy*. Presses de l'Université du Québec, 2018.

[94] S. Hallé and R. Villemaire, “Runtime monitoring of message-based workflows with data,” 10 2008, pp. 63–72.

[95] K. W. Hamlen, J. G. Morrisett, and F. B. Schneider, “Computability classes for enforcement mechanisms,” *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 1, pp. 175–205, 2006. [Online]. Available: https://doi.org/10.1145/1111596.1111601

[96] K. Havelund and G. Reger, “Runtime verification logics A language design perspective,” in *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, ser. Lecture Notes in Computer Science, L. Aceto, G. Bacci, G. Bacci, A. Ingólfsdóttir, A. Legay, and R. Mardare, Eds., vol. 10460. Springer, 2017, pp. 310–338.

[97] K. Havelund and G. Rosu, “Efficient monitoring of safety properties,” *Int. J. Softw. Tools Technol. Transf.*, vol. 6, no. 2, pp. 158–173, 2004.

[98] K. Havelund, D. A. Peled, and D. Ulus, “The dejavu runtime verification benchmark,” 2018.

[99] K. Havelund, G. Reger, D. Thoma, and E. Zalinescu, “Monitoring events that carry data,” in *Lectures on Runtime Verification - Introductory and Advanced Topics*, ser. Lecture Notes in Computer Science, E. Bartocci and Y. Falcone, Eds. Springer, 2018, vol. 10457, pp. 61–102.

[100] K. Havelund, G. Reger, and G. Rosu, “Runtime verification past experiences and future projections,” in *Computing and Software Science - State of the Art and Perspectives*, ser. Lecture Notes in Computer Science, B. Steffen and G. J. Woeginger, Eds. Springer, 2019, vol. 10000, pp. 532–562.

[101] L. Holík, M. Koreň, M. Novák, J. Šimáček, and J. Třmač, “Panda: Monitoring and diagnosis of distributed systems,” *arXiv preprint arXiv:1905.11953*, 2019.

[102] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas, “Attribute-based access control,” *Computer*, vol. 48, no. 2, pp. 85–88, 2015.

[103] J. Huang, C. Erdogan, Y. Zhang, B. M. Moore, Q. Luo, A. Sundaresan, and G. Rosu,

"ROSRV: runtime verification for robots," in *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, ser. Lecture Notes in Computer Science, B. Bonakdarpour and S. A. Smolka, Eds., vol. 8734. Springer, 2014, pp. 247–254.

[104] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok, "Software monitoring with controllable overhead," *Int. J. Softw. Tools Technol. Transf.*, vol. 14, no. 3, pp. 327–347, 2012.

[105] Y. Joshi, G. M. Tchamgoue, and S. Fischmeister, "Runtime verification of LTL on lossy traces," in *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, A. Seffah, B. Penzenstadler, C. Alves, and X. Peng, Eds. ACM, 2017, pp. 1379–1386.

[106] K. Kalajdzic, E. Bartocci, S. A. Smolka, S. D. Stoller, and R. Grosu, "Runtime verification with particle filtering," in *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, 2013, pp. 149–166.

[107] H. Kallwies, M. Leucker, and C. Sánchez, "Symbolic runtime verification for monitoring under uncertainties and assumptions," in *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings*, ser. Lecture Notes in Computer Science, A. Bouajjani, L. Holík, and Z. Wu, Eds., vol. 13505. Springer, 2022, pp. 117–134.

[108] H. Kallwies, M. Leucker, C. Sánchez, and T. Scheffel, "Anticipatory recurrent monitoring with uncertainty and assumptions," in *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*, ser. Lecture Notes in Computer Science, T. Dang and V. Stolz, Eds., vol. 13498. Springer, 2022, pp. 181–199.

[109] A. Khalid and L. C. Briand, "Checking data completeness in test data using runtime verification," in *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, 2015, pp. 276–290.

[110] R. Khoury and S. Hallé, "Runtime enforcement with partial control," in *Foundations*

and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers, ser. Lecture Notes in Computer Science, J. García-Alfaro, E. Kranakis, and G. Bonfante, Eds., vol. 9482. Springer, 2015, pp. 102–116. [Online]. Available: https://doi.org/10.1007/978-3-319-30303-1_7

[111] ——, "Tally keeping-LTL: An LTL semantics for quantitative evaluation of LTL specifications," in *2018 IEEE International Conference on Information Reuse and Integration, IRI 2018, Salt Lake City, UT, USA, July 6-9, 2018*. IEEE, 2018, pp. 495–502.

[112] R. Khoury and N. Tawbi, "Which security policies are enforceable by runtime monitors? a survey," *Computer Science Review*, vol. 6, no. 1, pp. 27–45, 2012.

[113] ——, "Corrective enforcement: A new paradigm of security policy enforcement by monitors," *ACM Transactions on Information and System Security*, vol. 15, no. 2, p. 10, 2012.

[114] R. Khoury, S. Hallé, and O. Waldmann, "Execution trace analysis using LTL-FO ˆ+," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 9953, 2016, pp. 356–362. [Online]. Available: https://doi.org/10.1007/978-3-319-47169-3_26

[115] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European conference on object-oriented programming*. Springer, 1997, pp. 220–242.

[116] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, "Computational analysis of run-time monitoring - fundamentals of java-mac," *Electron. Notes Theor. Comput. Sci.*, vol. 70, no. 4, pp. 80–94, 2002.

[117] M. A. Köhl, H. Hermanns, and S. Biewer, "Efficient monitoring of real driving emissions," in *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, ser. Lecture Notes in Computer Science, C. Colombo and M. Leucker, Eds., vol. 11237. Springer, 2018, pp. 299–315.

[118] S. Konur, "A survey on temporal logics for specifying and verifying real-time systems," *Frontiers Comput. Sci.*, vol. 7, no. 3, pp. 370–403, 2013.

[119] M. Kowalski, S. Hoffmann, and J. Halleux, "Umbral: A stream processing language for runtime verification of real-time systems," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2019, pp. 688–699.

[120] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real Time Syst.*, vol. 2, no. 4, pp. 255–299, 1990.

[121] K. G. Larsen, "Proof systems for satisfiability in hennessy-milner logic with recursion," *Theor. Comput. Sci.*, vol. 72, no. 2&3, pp. 265–288, 1990.

[122] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: efficient static binary instrumentation for linux," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*. IEEE Computer Society, 2010, pp. 175–183.

[123] O. Legunsen, D. Marinov, and G. Rosu, "Evolution-aware monitoring-oriented programming," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 615–618. [Online]. Available: https://doi.org/10.1109/ICSE.2015.206

[124] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebraic Methods Program.*, vol. 78, no. 5, pp. 293–303, 2009. [Online]. Available: https://doi.org/10.1016/j.jlap.2008.08.004

[125] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, and D. Thoma, "Runtime verification for timed event streams with partial information," in *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, 2019, pp. 273–291.

[126] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, 2017.

[127] J. Li, J. Lee, and L. Liao, "A novel algorithm for training hidden markov models with positive and negative examples," in *IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2020, Virtual Event, South Korea, December 16-19, 2020*, T. Park, Y. Cho, X. Hu, I. Yoo, H. G. Woo, J. Wang, J. C. Facelli, S. Nam, and M. Kang, Eds. IEEE, 2020, pp. 305–310.

[128] N. Li, "Discretionary access control," in *Encyclopedia of Cryptography and Security, 2nd Ed*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, pp. 353–356.

[129] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of nonsafety policies," *ACM Transactions on Information and System Security*, vol. 12, no. 3, Jan. 2009.

[130] H. Liu, M. Huang, I. Janghorban, P. Ghorbannezhad, and C. Yoo, "Faulty sensor detection, identification and reconstruction of indoor air quality measurements in a subway station," in *2011 11th International Conference on Control, Automation and Systems*, 2011, pp. 323–328.

[131] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 190–200.

[132] M. Lupp, "Extensible markup language," in *Encyclopedia of GIS*, S. Shekhar, H. Xiong, and X. Zhou, Eds. Springer, 2017, p. 583.

[133] M. Mammass and F. Ghadi, "An overview on access control models," *Int. J. Appl. Evol. Comput.*, vol. 6, no. 4, pp. 28–38, 2015.

[134] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[135] J. F. Marques and J. Bernardino, "Analysis of data anonymization techniques," in *Proceedings of the 12th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2020, Volume 2: KEOD, Budapest, Hungary, November 2-4, 2020*, D. Aveiro, J. L. G. Dietz, and J. Filipe, Eds. SCITEPRESS, 2020, pp. 235–241.

[136] F. Martinelli, I. Matteucci, and C. Morisset, “From qualitative to quantitative enforcement of security policy,” in *Computer Network Security - 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, MMM-ACNS 2012, St. Petersburg, Russia, October 17-19, 2012. Proceedings*, ser. Lecture Notes in Computer Science, I. V. Kotenko and V. A. Skormin, Eds., vol. 7531. Springer, 2012, pp. 22–35. [Online]. Available: https://doi.org/10.1007/978-3-642-33704-8_3

[137] G. H. Mealy, “A method for synthesizing sequential circuits,” *The Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[138] S. Mehta and V. Pandit, “A survey on sampling techniques and applications,” in *Proceedings of the 16th International Conference on Management of Data, 2010, Nagpur, India*, P. S. Kumar, S. Parthasarathy, and S. Godbole, Eds. Allied Publishers, 2010, p. 11.

[139] Microsoft, “How BizTalk Server implements host throttling,” 2017, https://docs.microsoft.com/en-us/biztalk/core/how-biztalk-server-implements-host-throttling, Accessed March 14th, 2020.

[140] P. Moosbrugger, K. Y. Rozier, and J. Schumann, “R2U2: monitoring and diagnosis of security threats for unmanned aerial systems,” *Formal Methods Syst. Des.*, vol. 51, no. 1, pp. 31–61, 2017.

[141] L. Moura, A. Sampaio, M. Souza, J. P. Feitosa, A. Oliveira, and R. Marinho, “Varan: A tool for runtime monitoring and verification of system software,” in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 503–504.

[142] A. Mrad, S. Ahmed, S. Hallé, and É. Beaudet, “Babeltrace: A collection of transducers for trace validation,” in *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Qadeer and S. Tasiran, Eds., vol. 7687. Springer, 2012, pp. 126–130.

[143] S. R. M. Oliveira and O. R. Zaïane, “Privacy preserving clustering by data transformation,” *J. Inf. Data Manag.*, vol. 1, no. 1, pp. 37–52, 2010.

[144] C. Olston, J. Jiang, and J. Widom, "Adaptive filters for continuous queries over distributed data streams," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, A. Y. Halevy, Z. G. Ives, and A. Doan, Eds. ACM, 2003, pp. 563–574.

[145] G. J. Pace, R. Pardo, and G. Schneider, "On the runtime enforcement of evolving privacy policies in online social networks," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 9953, 2016, pp. 407–412. [Online]. Available: https://doi.org/10.1007/978-3-319-47169-3_33

[146] N. Patel and S. Patel, "A study on data perturbation techniques in privacy preserving data mining," 2016.

[147] R. Pegoraro, R. B. Halima, K. Drira, K. Guennoun, and J. M. Rosário, "A framework for monitoring and runtime recovery of web service-based applications," in *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume ISAS-2, Barcelona, Spain, June 12-16, 2008*, J. Cordeiro and J. Filipe, Eds., 2008, pp. 201–206.

[148] M. Pezze and M. Young, "A survey of software testing techniques," *ACM Computing Surveys (CSUR)*, vol. 40, no. 4, pp. 1–92, 2008.

[149] J. Piechotta, D. Holling, R. Hähnle, and A. Podelski, "Online detection of multiple violations in requirements specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, 2018, pp. 82–93.

[150] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, A. Rollet, and O. L. Nguena-Timo, "Runtime enforcement of timed properties," in *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Qadeer and S. Tasiran, Eds., vol. 7687. Springer, 2012, pp. 229–244. [Online]. Available: https://doi.org/10.1007/978-3-642-35632-2_23

[151] S. Pinisetty, V. Preoteasa, S. Tripakis, T. Jéron, Y. Falcone, and H. Marchand, "Predictive runtime enforcement," *Formal Methods Syst. Des.*, vol. 51, no. 1, pp. 154–199, 2017. [Online]. Available: https://doi.org/10.1007/s10703-017-0271-1

[152] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. [Online]. Available: https://doi.org/10.1109/SFCS.1977.32

[153] A. Pnueli and A. Zaks, "PSL model checking and run-time verification via testers," in *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, ser. Lecture Notes in Computer Science, J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085. Springer, 2006, pp. 573–586.

[154] M. O. Rabin, "Probabilistic automata," *Information and Control*, vol. 6, no. 3, pp. 230–245, 1963.

[155] M. O. Rabin and D. S. Scott, "Finite automata and their decision problems," *IBM J. Res. Dev.*, vol. 3, no. 2, pp. 114–125, 1959.

[156] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[157] G. Reger and K. Havelund, "What is a trace? A runtime verification perspective," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 9953, 2016, pp. 339–355.

[158] G. Reger, H. C. Cruz, and D. E. Rydeheard, "MarQ: Monitoring at runtime with QEA," in *TACAS 2015*, ser. LNCS, C. Baier and C. Tinelli, Eds., vol. 9035. Springer, 2015, pp. 596–610.

[159] T. Revathi and D. Ramaraj, "Challenges and methods of data perturbation techniques," 2017.

[160] M. Roudjane, D. Rebaine, R. Khoury, and S. Hallé, “Predictive analytics for event stream processing,” in *23rd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2019, Paris, France, October 28-31, 2019*. IEEE, 2019, pp. 171–182. [Online]. Available: https://doi.org/10.1109/EDOC.2019.00029

[161] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstic, J. M. Lourenço, D. Nickovic, G. J. Pace, J. Rufino, J. Signoles, D. Traytel, and A. Weiss, “A survey of challenges for runtime verification from advanced application domains (beyond software),” *Formal Methods Syst. Des.*, vol. 54, no. 3, pp. 279–335, 2019.

[162] R. S. Sandhu and P. Samarati, “Access control: principle and practice,” *IEEE Communications Magazine*, vol. 32, no. 9, pp. 40–48, 1994.

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

[164] K. Selyunin, S. Jaksic, T. Nguyen, C. Reidl, U. Hafner, E. Bartocci, D. Nickovic, and R. Grosu, “Runtime monitoring with recovery of the sent communication protocol,” in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 336–355.

[165] F. A. Siddique, T. T. II, N. Brunelle, and K. Skadron, “Deterministic vs. non deterministic finite automata in automata processing,” *CoRR*, vol. abs/2210.10077, 2022.

[166] J. Simmonds, S. Ben-David, and M. Chechik, “Monitoring and recovery of web service applications,” in *The Smart Internet - Current Research and Future Applications*, ser. Lecture Notes in Computer Science, M. H. Chignell, J. R. Cordy, J. Ng, and Y. Yesha, Eds. Springer, 2010, vol. 6400, pp. 250–288.

[167] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok, “Runtime verification with state estimation,” in *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, 2011, pp. 193–207.

[168] R. Taleb, S. Hallé, and R. Khoury, “A modular runtime enforcement model using multi-traces,” in *Foundations and Practice of Security - 14th International Symposium,*

FPS 2021, Paris, France, December 7-10, 2021, Revised Selected Papers, ser. Lecture Notes in Computer Science, E. Aïmeur, M. Laurent, R. Yaich, B. Dupont, and J. García-Alfaro, Eds., vol. 13291. Springer, 2021, pp. 283–302. [Online]. Available: https://doi.org/10.1007/978-3-031-08147-7_19

[169] R. Taleb, R. Khoury, and S. Hallé, "Runtime verification under access restrictions," in *9th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2021, Madrid, Spain, May 17-21, 2021*. IEEE, 2021, pp. 31–41.

[170] R. Taleb, S. Hallé, and R. Khoury, "Benchmark measuring the overhead of runtime enforcement using multi-traces," Feb. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.5976001

[171] R. Taleb, R. Khoury, and S. Hallé, "A modular pipeline for enforcement of security properties at runtime," *Annals of Telecommunications.*, April 2023, in press.

[172] C. Talhi, N. Tawbi, and M. Debbabi, "Execution monitoring enforcement for limited-memory systems," in *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services, PST 2006, Markham, Ontario, Canada, October 30 - November 1, 2006*, ser. ACM International Conference Proceeding Series, vol. 380. ACM, 2006, p. 38. [Online]. Available: https://doi.org/10.1145/1501434.1501480

[173] N. Tatbul, "Load shedding," in *Encyclopedia of Database Systems, Second Edition*, L. Liu and M. T. Özsu, Eds. Springer, 2018.

[174] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," in *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, Eds. Morgan Kaufmann, 2003, pp. 309–320.

[175] A. Tavanaei and A. S. Maida, "Training a hidden markov model with a bayesian spiking neural network," *J. Signal Process. Syst.*, vol. 90, no. 2, pp. 211–220, 2018.

[176] B. M. Thuraisingham, "Mandatory access control," in *Encyclopedia of Database Systems, Second Edition*, L. Liu and M. T. Özsu, Eds. Springer, 2018.

[177] M. Tiger and F. Heintz, *International Journal of Approximate Reasoning*, vol. 119, pp. 325–352, 2020.

[178] S. J. Upadhyaya, “Mandatory access control,” in *Encyclopedia of Cryptography and Security, 2nd Ed*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, pp. 756–758.

[179] J. Vallet, A. Mrad, S. Hallé, and É. Beaudet, “The relational database engine: An efficient validator of temporal properties on event traces,” in *17th IEEE International Enterprise Distributed Object Computing Conference Workshops, EDOC Workshops, Vancouver, BC, Canada, September 9-13, 2013*, E. Bagheri, D. Gasevic, S. Hallé, M. Hatala, H. R. M. Nezhad, and M. Reichert, Eds. IEEE Computer Society, 2013, pp. 275–284.

[180] M. Y. Vardi, “Automatic verification of probabilistic concurrent finite-state systems,” *Distributed Computing*, vol. 11, no. 3, pp. 139–155, 1998.

[181] M. Y. Vardi and P. Wolper, “Automata-theoretic techniques for modal logics of programs,” *Journal of computer and system sciences*, vol. 32, no. 2, pp. 183–221, 1986.

[182] S. Varvaressos, K. Lavoie, S. Gaboury, and S. Hallé, “Automated bug finding in video games: A case study for runtime monitoring,” *Comput. Entertain.*, vol. 15, no. 1, pp. 1:1–1:28, 2017. [Online]. Available: https://doi.org/10.1145/2700529

[183] M. Vella, C. Colombo, R. Abela, and P. Špaček, “RV-TEE: secure cryptographic protocol execution based on runtime verification,” *J. Comput. Virol. Hacking Tech.*, vol. 17, pp. 229–248, 2021.

[184] S. Wang, A. Ayoub, O. Sokolsky, and I. Lee, “Runtime verification of traces under recording uncertainty,” in *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, 2011, pp. 442–456.

[185] C. M. Wilcox and B. C. Williams, “Runtime verification of stochastic, faulty systems,” in *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, ser. Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and

N. Tillmann, Eds., vol. 6418. Springer, 2010, pp. 452–459.

[186] R. L. Wilson and P. A. Rosen, “Protecting data through perturbation techniques: The impact on knowledge discovery in databases,” *J. Database Manag.*, vol. 14, no. 2, pp. 14–26, 2003.

[187] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, “Detecting large-scale system problems by mining console logs,” in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, 2009, pp. 117–132.

[188] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, “SherLog: error diagnosis by connecting clues from run-time logs,” in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, 2010, pp. 143–154.

[189] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, “Be conservative: Enhancing failure diagnosis with proactive logging,” in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 293–306.

[190] Y. Zhang and J. B. D. Joshi, “Role-based access control,” in *Encyclopedia of Database Systems, Second Edition*, L. Liu and M. T. Özsu, Eds. Springer, 2018.

[191] G. Zhou, C. Yang, P. Lu, and X. Chen, “Runtime verification in uncertain environment based on probabilistic model learning,” *Mathematical Biosciences and Engineering*, vol. 19, no. 12, pp. 13 607–13 627, 2022.