Technical Report:
Correct Audit Logging in Hybrid-Dynamic Systems

Sepehr Amir-Mohammadian

Dept. of Computer Science,
University of the Pacific, Stockton, CA

April 2024

Abstract

Hybrid-dynamic models establish the foundational framework necessary for examining the expanding field of cyber-physical systems, emphasizing the integration of discrete computational steps with continuous physical dynamics. The widespread presence of these systems necessitates a reliable assurance mechanism for audit logging across various discrete and continuous components. This report explores an implementation model for these systems. We introduce an algorithm designed to equip such systems in accordance with a formal specification of audit logging requirements, which ensures the generation of accurate audit logs in any instrumented system.

Keywords— Audit logging, Cyber-physical systems, Programming languages, Security

1 Introduction

Audit logs play a crucial role in enhancing security across various domains by providing a detailed record of system activities and events. They serve as an indispensable source of information for detecting and investigating security incidents, breaches, and unauthorized access attempts [9]. By capturing essential details such as user actions, system configurations, network traffic, and application activities, audit logs enable security teams to monitor, analyze, and respond to potential threats effectively. Moreover, audit logs serve as a deterrent to malicious actors, as the knowledge of being monitored can dissuade unauthorized activities.

In cyber physical systems (CPSs), where the integration of physical processes with computing and networking capabilities is prevalent, audit logging becomes even more critical. These systems are subject to various security threats, including cyber attacks

targeting critical infrastructure [22], industrial control systems [2] and autonomous vehicles (AVs) [16]. Audit logging in CPSs enables the tracking of interactions between physical and digital components, which provides insights into system behavior, anomalies, and potential vulnerabilities. For instance, in smart grid systems, audit logs can help detect and prevent unauthorized access to energy distribution networks, ensuring the reliability and integrity of the grid [18]. Similarly, in AVs, audit logs are essential for recording sensor data, decision-making processes, and helping with the forensic analysis and liability attribution in case of accidents or cyber attacks [10].

In recent years, information-algebraic models [17], have emerged as valuable semantic frameworks for audit logging. These models interpret audit logs and the runtime structure of processes as information-algebraic elements, providing intuitive insights into their information content. The reliability of audit logging hinges on this algebraic interpretation, which compares the information content of audit logs with that of the program at runtime. Leveraging this semantic framework, an implementation model has been proposed for linear [5] and concurrent [6] computations, ensuring accurate audit logging through instrumentation techniques.

In this report, we explore the application of the aforementioned information-algebraic framework in CPSs. To this end, we use a programming linguistic model of CPSs, known as hybrid programs (HPs) [24, 25, 26]. Hybridity in the context of HPs refers to the simultaneous presence of discrete computational elements and continuous physical dynamics inherent in CPSs. HPs serve as a fundamental programming language for CPSs, allowing for the specification of integrated discrete and continuous behaviors. We use a variant of HPs, where the semantics is specified operationally (rather than denotationally) to facilitate the specification of audit log generation at runtime. Our formalism provides a model for developing CPS tools with correctness guarantees. This language model offers the following features.

- The language model supports the hybrid nature of CPSs, accommodating both discrete computational steps and continuous physical behavior.
- Leveraging a variant of HPs, our approach benefits from concise syntax and semantics. This facilitates the description of a broad spectrum of hybrid-dynamic systems.
- In order to articulate auditing requirements effectively, our model incorporates timestamps as part of the runtime environment. This enables the specification of the ordering of significant events.
- Fundamental to specifying auditing requirements is the abstraction of secure operations. Named functions serve as these fundamental units. They offer a versatile tool for expressing auditing needs across different languages and systems.

Utilizing the formalism with the aforementioned features empowers us to model CPSs that ensure the accurate generation of audit logs in line with the developed semantic framework. In this report, we present an instrumentation algorithm designed

to modify an HP based on precise audit logging requirements. We demonstrate the correctness of this algorithm, as per the semantic framework, which ensures that that the instrumented concurrent system produces accurate audit logs. The implementation of audit logging policies through code instrumentation offers a separation of policy from code, which lays the groundwork for studying the effectiveness of enforcement mechanisms using formal methods. Additionally, it can be automatically applied to legacy code to bolster system accountability.

1.1 An Illustrative Example

Let's consider a simplified scenario to illustrate the necessity of studying correct audit logging in CPSs. We will revisit this example multiple times throughout the report. Our example is an AV system that includes a controller and a physical machinery, and the task is to log a significant event like *hard braking*. Here is how such a sequence of events might unfold, culminating in a log entry:

- The controller continuously monitors the vehicle's speed and uses sensors to detect objects in its path. For instance, as the vehicle cruises at 60 mph, a pedestrian unexpectedly steps onto the road.
- The detection of a pedestrian in the vehicle's path triggers the controller's obstacle avoidance algorithm. The controller assesses the distance and speed relative to the obstacle. It calculates that normal braking will not suffice to avoid a collision.
- Based on the controller's decision, a command is sent to the physical machinery to execute a hard brake.
- The physical machinery applies the vehicle's brakes forcefully and quickly to reduce speed dramatically in a short time.

As the hard braking occurs, this critical action is captured as a log event. The system records this event along with pertinent details such as the vehicle's speed at the time of braking, the time, and the location. Retroactively, the log can be reviewed for compliance with safety protocols or used for improving the decision-making algorithm based on real-world outcomes. In this scenario, the sequence of detection, decision, execution, and logging is crucial for ensuring both the immediate safety of the vehicle's occupants and others on the road, as well as for long-term improvements and accountability in AV operations.

1.2 Report Outline

The rest of the technical report is organized as follows: In Section 2, we review the information-algebraic semantic framework for correct audit logging. In Section 3, we explore the implementation model for HPs by specifying the source and target

language models, as well as the instrumentation algorithm that maps an HP from the source language to an HP in the target language. In addition, towards the end of this section the main results of the work are specified. Section 4 discusses the related work. Finally, Section 5 concludes the report.

2 Semantics of Audit Logging

In this section, we explore the information-algebraic semantics of audit logging. The content of this section has originally been explored elsewhere [5], but we delve into it here to provide an standalone formal presentation. Minor adjustments have been made to the model to accommodate nondeterministic runtime behavior inherent in hybrid-dynamic systems.

2.1 Information-Algebraic Semantic Framework

To specify the generation of audit logs during runtime, we must abstract program states and trace their evolution through computation. A program configuration, denoted by κ, serves as a representation of the program's state at any given moment during execution. We establish a binary reduction relation among configurations, $\kappa \longrightarrow \kappa'$, to signify computational steps. A program trace, τ, constitutes a potentially infinite sequence of program configurations, depicted as $\tau = \kappa_0\kappa_1 \cdots$, where κ_i denotes the ith configuration in the sequence and $\kappa_i \longrightarrow \kappa_{i+1}$. The set encompassing all traces is denoted by $\mathcal{T}$, while the $\mathit{prefix}(\tau)$ is defined as the set containing all initial segments of τ.

Information algebra is employed to establish the concept of correctness for audit logs. In Section 2.2, we apply this abstract algebraic framework to represent a particular set of audit logging criteria.

Definition 2.1 (Information algebra)**.** An information algebra (Φ, Ψ) comprises a two-sorted algebra, consisting of an Abelian semigroup of information elements denoted as Φ, alongside a lattice of querying domains denoted as Ψ. Within this algebra, two fundamental operators are presumed: a combination operator $(\otimes) : \Phi \times \Phi \to \Phi$, and a focusing operator $(^{\Rightarrow}) : \Phi \times \Psi \to \Phi$.

An information algebra adheres to a set of properties [17] concerning these combination and focusing operators. Throughout, we use X, Y, etc., to denote elements of Φ, and E to denote elements of Ψ.

In the information algebra (Φ, Ψ), let $X, Y \in \Phi$ represent information elements that can be combined to form a more comprehensive information element $X \otimes Y$. $E \in \Psi$ signifies a querying domain with a specific level of granularity utilized by the focusing operator to extract information from an information element X, denoted as $X^{\Rightarrow E}$.

For instance, relational algebra serves as an example of information algebra, where relations serve as instances of information elements, sets of attributes represent querying domains, the natural join of two relations defines the combination operator, and the projection of a relation onto a set of attributes defines the focusing operator [17].

The combination of information elements gives rise to a partial order relation, $\preccurlyeq$, among these elements, defined as follows: $X \preccurlyeq Y$ iff $X \otimes Y = Y$. In simpler terms, $X \preccurlyeq Y$ indicates that Y encompasses the information element X.

In the context of audit logging semantics, execution traces are regarded as information elements, representing the information content of the trace. We define $\lfloor \cdot \rfloor : \mathcal{T} \to \Phi$ as a mapping where $\lfloor \tau \rfloor$ denotes the information content of the trace τ. We require $\lfloor \cdot \rfloor$ to be injective and monotonically increasing, such that if $\tau' \in \mathit{prefix}(\tau)$ then $\lfloor \tau' \rfloor \preccurlyeq \lfloor \tau \rfloor$, ensuring that longer execution traces contain more information.

In the subsequent definition, we establish audit logging requirements in an abstract manner, referring to this abstraction as a *logging specification*. This definition is sufficiently abstract to accommodate various execution models and representations of information. In Sections 2.2 and 3.2, we provide concrete instantiations of this definition, offering guidance on implementing audit logging requirements.

Definition 2.2 (Logging specifications)**.** The logging specification LS is characterized as a mapping from program traces to information elements, expressed as $LS : \mathcal{T} \to \Phi$. Essentially, $LS(\tau)$ indicates what information should be recorded if the program follows the execution trace τ.

It is important to note that although $\lfloor \cdot \rfloor$ and LS share the same signature—mapping from traces to information elements—they serve different conceptual purposes. $\lfloor \tau \rfloor$ represents all the information within τ, while $LS(\tau)$ signifies the specific information intended for recording in the log if the program follows the execution trace τ.

We denote an audit log as $\mathbb{L}$, representing a set of data gathered at runtime. Let $\mathcal{L}$ denote the set of all audit logs. Evaluating the correctness of an audit log involves comparing its information content with that of the trace that generated it. Thus, we define a mapping that yields the information content of an audit log. We abuse the notation by considering $\lfloor \cdot \rfloor : \mathcal{L} \to \Phi$ as this mapping. Consequently, $\lfloor \mathbb{L} \rfloor$ denotes the information content of the audit log $\mathbb{L}$. We assume that $\lfloor \cdot \rfloor$ applied to audit logs is injective and monotonically increasing; in other words, if $\mathbb{L} \subseteq \mathbb{L}'$, then $\lfloor \mathbb{L} \rfloor \preccurlyeq \lfloor \mathbb{L}' \rfloor$. Thus, a more inclusive audit log contains more information.

The concept of correct audit logging can be established by considering an execution trace and a logging specification. This involves comparing the information content of the audit log with the information that the logging specification mandates to be recorded in the log, given the execution trace. The following definition formalizes this relationship.

Definition 2.3 (Correctness of audit logs)**.** An audit log $\mathbb{L}$ is deemed *correct* with respect to a logging specification LS and a program trace τ if both $\lfloor \mathbb{L} \rfloor \preccurlyeq LS(\tau)$ and $LS(\tau) \preccurlyeq \lfloor \mathbb{L} \rfloor$ are satisfied. The former indicates the necessity of the information in the audit log, while the latter signifies the sufficiency of that information.

In a runtime system generating audit logs, the stored logs become integral to its configuration. We define $\mathit{logof}(\kappa)$ to represent the residual log of the program configuration κ, which encompasses all recorded audit logs in κ. It is reasonable to expect that the residual log within configurations expands with the progression of execution. Consequently, the residual log of a trace is defined utilizing logof.

Definition 2.4 (Residual log of a program trace)**.** The residual log of a finite program trace τ is $\mathbb{L}$, denoted by $\tau \rightsquigarrow \mathbb{L}$, iff $\tau = \kappa_0\kappa_1 \cdots \kappa_n$ and $\mathit{logof}(\kappa_n) = \mathbb{L}$.

Note that if $\tau \rightsquigarrow \mathbb{L}$, $\mathbb{L}$ may not necessarily be correct with respect to a given LS and τ. If the residual log of a trace remains correct throughout execution, the trace is termed *ideally-instrumented*. A program trace τ is ideally instrumented for a logging specification LS if, for any trace τ' and audit log $\mathbb{L}$, $\tau' \in \mathit{prefix}(\tau)$ and $\tau' \rightsquigarrow \mathbb{L}$ imply correctness of $\mathbb{L}$ with respect to τ' and LS. Audit logging constitutes an enforceable security property on an execution trace [5].

Let $\mathfrak{p}$ denote a program with operational semantics. We say $\mathfrak{p} \Downarrow \tau$ if $\mathfrak{p}$ can produce trace τ', either deterministically or non-deterministically, and $\tau \in \mathit{prefix}(\tau')$. We extend this notation to configurations, using $\kappa \Downarrow \tau$ to convey the same concept for configuration κ.

Employing program instrumentation techniques, we enforce a logging specification on a program. An *instrumentation algorithm* takes the program and logging specification as inputs, instrumenting the program with audit logging capabilities to generate the required log. Formally, an instrumentation algorithm is a partial function $\mathcal{I}(\mathfrak{p}, LS)$, which instruments $\mathfrak{p}$ according to LS to generate audit logs appropriate for LS. Here, $\mathfrak{p}$ is termed the *source program*, and the resulting instrumented program, $\mathcal{I}(\mathfrak{p}, LS)$, is termed the *target program*. Source and target traces refer to the traces of the source and target programs, respectively.

We naturally expect that the instrumentation algorithm does not drastically alter the semantics of the original program. The target program should behave similarly to the source program, with differences primarily in operations related to audit logging. We term this attribute of an instrumentation algorithm *semantics preservation* and define it as follows. This definition is sufficiently abstract to accommodate various source and target programs (with different runtime semantics) and instrumentation techniques. It relies on a binary relation $:\approx$, referred to as the *correspondence relation*, which connects the source and target traces. Depending on the implementations of the source and target programs and the instrumentation algorithm, the correspondence relation can be defined accordingly.

Definition 2.5 (Semantics preservation by the instrumentation algorithm)**.** The instrumentation algorithm $\mathcal{I}$ is semantics-preserving if, for all programs $\mathfrak{p}$ and logging specifications LS, where $\mathcal{I}(\mathfrak{p}, LS)$ is defined, the following conditions hold: 1) For any trace τ, if $\mathfrak{p} \Downarrow \tau$, then there exists some trace τ' such that $\mathcal{I}(\mathfrak{p}, LS) \Downarrow \tau'$ and $\tau :\approx \tau'$, and 2) For any trace τ, if $\mathcal{I}(\mathfrak{p}, LS) \Downarrow \tau$, then there exists some trace τ' such that $\mathfrak{p} \Downarrow \tau'$, and $\tau' :\approx \tau$.

The crucial aspect of an instrumentation algorithm is the quality of audit logs generated by the instrumented program. The information-algebraic semantic framework offers a platform to define correct instrumentation algorithms for audit logging purposes. Let $\mathfrak{p}$ be a target program, and τ be a source trace. The simulated logs of τ by $\mathfrak{p}$ constitute the set $\mathit{simlogs}(\mathfrak{p}, \tau)$, defined as $\mathit{simlogs}(\mathfrak{p}, \tau) = \{\mathbb{L} \mid \exists \tau'.\mathfrak{p} \Downarrow \tau' \wedge \tau :\approx \tau' \wedge \tau' \rightsquigarrow \mathbb{L}\}$. Using this set, we can straightforwardly define the correctness of instrumentation algorithms. Intuitively, the instrumentation algorithm $\mathcal{I}$ is correct if the instrumented program generates audit logs that are correct with respect to the logging specification and the source trace. This criterion must hold for any source program, any logging specification, and any possible log generated by the instrumented program.

Definition 2.6 (Correctness of the instrumentation algorithm)**.** The instrumentation algorithm $\mathcal{I}$ is deemed correct if, for all source programs $\mathfrak{p}$, traces τ, and logging specifications LS, $\mathfrak{p} \Downarrow \tau$ implies that for any $\mathbb{L} \in \mathit{simlogs}(\mathcal{I}(\mathfrak{p}, LS), \tau)$, $\mathbb{L}$ is correct with respect to LS and τ.

2.2 Instantiation of Logging Specification

Definition 2.2 abstractly defines a logging specification as a mapping from program traces to information elements. To apply this definition effectively in real-world scenarios, we must instantiate it with suitable structures, particularly utilizing information algebra (Definition 2.1). We prioritize logical specification for audit logging requirements due to its ease of use, expressive power, well-understood semantics, and compatibility with logic programming engines, such as Horn clause logic for subsets of first-order logic (FOL). In this section, we instantiate information algebra with FOL, which adequately captures computational events and temporal relations among them. Nonetheless, alternative logic variants may also be explored for this purpose.

To instantiate information algebra, we must specify the contents of the set of information elements Φ and the lattice of querying domains Ψ, along with the definitions of combination and focusing operators. Definitions 2.7, 2.8, and 2.9 achieve these instantiations.

Definition 2.7 provides an instantiation of an FOL-based set of information elements. In this instantiation, an information element corresponds to a closed set of FOL formulas, established under a proof-theoretic deductive system.

Definition 2.7 (Set of closed sets of FOL formulas)**.** We let φ range over FOL formulas and Γ over sets of FOL formulas. The judgment $\Gamma \vdash \varphi$ signifies a derivation obtained through a sound and complete natural deduction proof theory of FOL. We define the closure operation *Closure* as $\mathit{Closure}(\Gamma) = \{\varphi \mid \Gamma \vdash \varphi\}$. Consequently, the set of closed sets of FOL formulas is denoted as $\Phi_{FOL} = \{\Gamma \mid \Gamma = \mathit{Closure}(\Gamma)\}$.

Definition 2.8 introduces the lattice of querying domains for the FOL-based information algebra. Here, a query domain constitutes a subset of FOL, defined over specific predicate symbols.

Definition 2.8 (Lattice of FOL sublanguages)**.** Let *Preds* represent the set of all assumed predicate symbols along with their arities. If $S \subseteq \mathit{Preds}$, we denote the sublanguage $\mathit{FOL}(S)$ as the collection of well-formed FOL formulas over predicate symbols in S. The set of all such sublanguages $\Psi_{\mathit{FOL}} = \{\mathit{FOL}(S) \mid S \subseteq \mathit{Preds}\}$ forms a lattice induced by the set containment relation.

Finally, Definition 2.9 provides the instantiation of the combination and focusing operators for the FOL-based information algebra. Combination involves taking the closure of the union of two sets of formulas. Focusing entails taking the closure of the intersection of an information element and a query domain.

Definition 2.9 (Combination and focusing in $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$)**.** Let $(\otimes) : \Phi_{\mathit{FOL}} \times \Phi_{\mathit{FOL}} \to \Phi_{\mathit{FOL}}$ be defined as $\Gamma \otimes \Gamma' = \mathit{Closure}(\Gamma \cup \Gamma')$, and $(^{\Rightarrow}) : \Phi_{\mathit{FOL}} \times \Psi_{\mathit{FOL}} \to \Phi_{\mathit{FOL}}$ be defined as $\Gamma^{\Rightarrow \mathit{FOL}(S)} = \mathit{Closure}(\Gamma \cap \mathit{FOL}(S))$.

The pair $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$ forms an information algebra, as defined in Definitions 2.7, 2.8, and 2.9. The detailed proof can be found elsewhere [4]. To utilize $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$ as a framework for audit logging, we also need to instantiate the mapping $\lfloor \cdot \rfloor$. This mapping interprets both execution traces and audit logs as information elements.

Definition 2.10 (Mapping traces and audit logs to information elements in $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$)**.** We define $\mathit{toFOL}(\cdot) : (\mathcal{T} \cup \mathcal{L}) \to \mathit{FOL}(\mathit{Preds})$ as an injective and monotonically increasing function. Then, to interpret both traces and logs as information elements in $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$, we instantiate $\lfloor \cdot \rfloor = \mathit{Closure}(\mathit{toFOL}(\cdot))$.

Now, we can instantiate a logging specification LS in the information algebra $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$. To do this, we assume a set of audit logging rules and definitions given in FOL, denoted by Γ. Additionally, we assume a set of predicate symbols, denoted by S, which represent the predicates whose derivations need to be logged at runtime. In this setting, a logging specification combines the information content of a trace τ with the closure of Γ and then focuses on the predicates specified in S. Intuitively, given Γ and S, a logging specification maps a trace τ to the set of all predicates whose symbols are in S and are derivable based on the rules in Γ and the events in τ.

Definition 2.11 (Logging Specification in $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$)**.** For a set of FOL formulas Γ and a subset of predicate symbols $S \subseteq \mathit{Preds}$, a logging specification $\mathit{spec}(\Gamma, S) : \mathcal{T} \to \Phi_{\mathit{FOL}}$ is defined as $\mathit{spec}(\Gamma, S) = \tau \mapsto (\lfloor \tau \rfloor \otimes \mathit{Closure}(\Gamma))^{\Rightarrow \mathit{FOL}(S)}$.

3 Implementation Model on Hybrid-Dynamic Systems

This section introduces an implementation model aimed at ensuring accurate audit logging in hybrid-dynamic systems. We employ HPs to define the hybrid-dynamic system and put forth an instrumentation algorithm that modifies the program according

to a given logging specification. Furthermore, we detail and establish pertinent properties, notably including the correctness of the instrumentation algorithm (Definition 2.6).

In Section 3.1, we discuss the syntax and semantics of the source program model. Section 3.2 introduces a class of logging specifications adept at specifying temporal relations among computational events in hybrid-dynamic systems. Following this, Section 3.3 outlines the syntax and semantics of hybrid programs enriched with audit logging capabilities. Finally, in Section 3.4, we delve into the instrumentation algorithm and elucidate the properties it adheres.

3.1 Source HP Model

We consider HPs as our source language model, refered by $\mathcal{HP}$. Syntax and semantics of HPs are defined in the following.

3.1.1 Syntax and Semantics of HPs

The syntax and semantics of HPs [26] are grounded in real arithmetic polynomial terms and FOL of real arithmetic.

Polynomial terms We define polynomial terms e with rational coefficients over a countably infinite set of variables $\mathcal{V}$. A polynomial term e takes the form $e ::= x \mid c \mid e.e \mid e + e$, where $x \in \mathcal{V}$ and $c \in \mathbb{Q}$ (a rational number). A State, ω, is a mapping from variables to real numbers, i.e., $\omega : \mathcal{V} \to \mathbb{R}$. We let $r \in \mathbb{R}$ to range over real numbers throughout the report. The semantics of a polynomial term e is determined by a state, $\omega[\![e]\!]$, given in Figure 1.

FOL of real arithmetic The FOL of real arithmetic is syntactically defined by $P ::= e = e \mid e \geq e \mid \neg P \mid P \wedge P \mid \exists x P$. Additional syntactic structures, such as disjunction, implication, universal quantification, etc., can be defined using this minimal syntax. A state ω models a predicate P, denoted by $\omega \vDash P$, as per the definition in Figure 1.

HPs : HPs are defined syntactically as follows: $\alpha ::= x := e \mid x := * \mid f(\bar{e}) \mid ?P \mid x' = e \;\&\; P \mid \alpha \cup \alpha \mid \alpha; \alpha \mid \alpha^*$. The statement $x := e$ updates x to the value of e. $x := *$ updates x to a nondeterministic value. $f(\bar{e})$ refers to invoking function f with inputs $\bar{e}$. A codebase $\mathcal{C}$ maps function names to their definitions: $f(\bar{x}) = \alpha$. The statement $?P$ is a boolean test. $x' = e \;\&\; P$ is a continuous program with an ordinary differential equation $x' = e$ and an evolution domain P, where x' denotes the time derivative of x. Continuous programs are restricted to polynomial differential equations. We may specify a vector of such equations to specify a continuous evolution. We use $\alpha \cup \alpha$ for nondeterministic choice, $\alpha; \alpha$ for sequencing, and α^* for iterating α nondeterministic number of times. We show empty sequence of HPs with ϵ.

$$\omega[\![x]\!] = \omega(x) \qquad \omega[\![c]\!] = c \qquad \omega[\![e.e']\!] = \omega[\![e]\!].\omega[\![e']\!] \qquad \omega[\![e + e']\!] = \omega[\![e]\!] + \omega[\![e']\!]$$

$$\frac{\omega[\![e]\!] = [\![e']\!]}{\omega \vDash e = e'} \qquad \frac{\omega[\![e]\!] \geq [\![e']\!]}{\omega \vDash e \geq e'} \qquad \frac{\omega \nvDash P}{\omega \vDash \neg P} \qquad \frac{\omega \vDash P \qquad \omega \vDash P'}{\omega \vDash P \wedge P'}$$

$$\frac{\exists r \in \mathbb{R}.\omega[x \mapsto r] \vDash P}{\omega \vDash \exists x P} \qquad \text{O1}\quad (t, \omega, x := e) \longrightarrow (t + 1, \omega[x \mapsto \omega[\![e]\!]], \epsilon)$$

$$\text{O2}\quad \frac{r \in \mathbb{R}}{(t, \omega, x := *) \longrightarrow (t + 1, \omega[x \mapsto r], \epsilon)} \qquad \text{O3}\quad \frac{\omega \vDash P}{(t, \omega, ?P) \longrightarrow (t + 1, \omega, \epsilon)}$$

$$\text{O4}\quad \frac{\mathcal{C}(f) = [f(\bar{x}) = \alpha] \qquad \omega[\![\bar{e}]\!] = \bar{r}}{(t, \omega, f(\bar{e})) \longrightarrow (t, \omega[\bar{x} \mapsto \bar{r}], \alpha)}$$

$$\text{O5}\quad \frac{\exists r \geq 0, \varphi : [0, r] \to \mathcal{S}. \qquad \big((\varphi \text{ solves } x' = e \text{ on } [0, r]) \wedge (\forall t \in [0, r].\varphi(t) \vDash P)\big)}{(t, \varphi(0), x' = e \;\&\; P) \longrightarrow (t + r, \varphi(r), \epsilon)}$$

$$\text{O6}\quad \frac{i = 1, 2}{(t, \omega, \alpha_1 \cup \alpha_2) \longrightarrow (t + 1, \omega, \alpha_i)} \qquad \text{O7}\quad \frac{(t, \omega, \alpha) \longrightarrow (t', \omega', \alpha')}{(t, \omega, \mathcal{E}[\alpha]) \longrightarrow (t', \omega', \mathcal{E}[\alpha'])}$$

$$\text{O8}\quad \frac{\alpha_1 \equiv \alpha_1' \qquad \alpha_2 \equiv \alpha_2' \qquad (t, \omega, \alpha_1) \longrightarrow (t', \omega', \alpha_2)}{(t, \omega, \alpha_1') \longrightarrow (t', \omega', \alpha_2')}$$

Figure 1: Semantics of 1) polynomial terms: $\omega[\![e]\!]$, 2) FOL of real arithmetic: $\omega \vDash P$, and 3) HPs: $\kappa \longrightarrow \kappa'$.

We define two HPs α_1 and α_2 structurally congruent, $\alpha_1 \equiv \alpha_2$ according to the following rules: i) Structural congruence is an equivalence relation, ii) $\epsilon; \alpha \equiv \alpha$, iii) $\alpha^* \equiv \alpha^0 \cup \alpha^1 \cup \cdots = \cup_{i \in \mathbb{N}} \alpha^i$, where α^i is the iteration α for i times, defined as $\alpha^0 = ?\top$ and $\alpha^{i+1} = \alpha^i; \alpha$, and iv) $\alpha_1 \equiv \alpha_2$ implies $\mathcal{E}[\alpha_1] \equiv \mathcal{E}[\alpha_2]$, where $\mathcal{E}$ is the evaluation context and is defined as $\mathcal{E} ::= [\,] \mid \mathcal{E}; \alpha \mid \epsilon; \mathcal{E}$.

Having defined HPs syntactically, we define a program as an HP $\alpha_{\mathfrak{p}}$ along with the codebase of all defined functions, i.e., $\mathfrak{p} = \langle \alpha_{\mathfrak{p}}, \mathcal{C} \rangle$. We assume that $\alpha_{\mathfrak{p}}$ simply invokes the specific function *main*().

We define $\kappa ::= (t, \omega, \alpha)$ where $t \in \mathbb{R}^{\geq 0}$ is a timestamp. Accordingly, we define the timestamped operational semantics of HPs in Figure 1. According to rule O1, $x := e$ updates the state ω by mapping x to the value of e in ω. In rule O2, ω is updated with x being mapped to a nondeterministic real value. $?P$ is defined only for states that model P, without altering the state, according to rule O3. In rule O4, invoking f with inputs $\bar{e}$ runs the body of the function in a new state, where fresh parameter names are mapped to their corresponding polynomial term values. According to rule O5, $x' = e \;\&\; P$ is executable if there exists a solution for the equation $x' = e$ in the domain P. In this case, x is updated according to $x' = e$ within some nondeterministic continuous time span $[0, r]$, where every state update must satisfy the domain condition P. Rule O6 describes how $\alpha_1 \cup \alpha_2$ nondeterministically chooses between α_1 and α_2 to run. Rule O7 specifies reduction of sequences of HPs using an evaluation context, and rule O8 states that reduction follows structural congruence.

Example 3.1. Having introduced HPs as the source language model, let's revisit the AV example described in Section 1.1. We can describe the AV system as the HP $(\mathit{controller}; \mathit{machine})^*$, where *controller* is the computational component and *machine* is the physical component of the AV (the engine, braking system, etc.) defined in Figure 2. Function *sen*() retrieves data from speed sensors and environmental sensors to detect obstacles. Predicate P_1 returns true if an obstacle is detected. Function *assess*() analyzes the data to determine the threat level and appropriate response, e.g., braking level or maneuver. Predicate P_2 returns true if the threat level is high enough for hard breaking. Function *cmdHB*() executes the hard braking command. In our language model, the command can be described by simply setting the acceleration parameter to a proper negative value. Function *init*() assigns certain initial values for physical evolution, e.g., time and acceleration (effectively coming from *cmdHB*()). Finally, the function *evolve*() delineates the temporal progression of the AV's position through the application of differential equations. This function is formally expressed as $k' = 1, x' = v, v' = a \;\&\; k \leq p$, where k, x, v, and a represent the time, position, velocity, and acceleration of the AV, respectively. Here, k is a continuously increasing time variable. The evolution of the AV's position, velocity, and timer is confined within its evolution domain, which imposes an upper limit on the driving duration—denoted by $k \leq p$. This constraint ensures that the AV does not operate continuously beyond a pre-set limit of p time units for safety reasons, after which control is transferred back to the *controller*.

$$controller = \Big(sen(); \big(?P_1; assess(); (?P_2; cmdHB()) \ \cup \ ?\neg P_2\big) \ \cup \ ?\neg P_1\Big)^*$$

$$machine = init(); evolve()$$

Figure 2: The definition of an AV as an HP in Example 3.1.

$$\frac{\kappa = (t, \omega, \alpha) \qquad \alpha \equiv \mathcal{E}[x := e]}{toFOL(\kappa) = \{\text{DAssign}(t, x, e), \text{Context}(t, \mathcal{E}), \text{State}(t, \omega)\}}$$

$$\frac{\kappa = (t, \omega, \alpha) \qquad \alpha \equiv \mathcal{E}[x := *]}{toFOL(\kappa) = \{\text{NDAssign}(t, x), \text{Context}(t, \mathcal{E}), \text{State}(t, \omega)\}}$$

$$\frac{\kappa = (t, \omega, \alpha) \qquad \alpha \equiv \mathcal{E}[?P]}{toFOL(\kappa) = \{\text{Test}(t, P), \text{Context}(t, \mathcal{E}), \text{State}(t, \omega)\}}$$

$$\frac{\kappa = (t, \omega, \alpha) \qquad \alpha \equiv \mathcal{E}[f(\bar{e})]}{toFOL(\kappa) = \{\text{Call}(t, f, \bar{e}), \text{Context}(t, \mathcal{E}), \text{State}(t, \omega)\}}$$

$$\frac{\kappa = (t, \omega, \alpha) \qquad \alpha \equiv \mathcal{E}[x' = e \ \& \ P]}{toFOL(\kappa) = \{\text{Continuum}(t, x' = e, P), \text{Context}(t, \mathcal{E}), \text{State}(t, \omega)\}}$$

$$\frac{\kappa = (t, \omega, \alpha) \qquad \alpha \equiv \mathcal{E}[\alpha_1 \cup \alpha_2]}{toFOL(\kappa) = \{\text{Choice}(t, \alpha_1, \alpha_2), \text{Context}(t, \mathcal{E}), \text{State}(t, \omega)\}}$$

Figure 3: Instantiation of $toFOL(\cdot)$.

3.1.2 Instantiation of $toFOL(\cdot)$

To logically specify a trace, we must instantiate $toFOL(\cdot)$ according to Definition 2.10. We consider the following predicates to logically specify a trace: DAssign/3, NDAssign/2, Test/2, Call/3, Continuum/3, Choice/3, State/2, and Context/2. Note that $/n$ refers to the arity of the predicate.

We define a function to logically specify a configuration within a trace. For this purpose, we introduce the helper function $toFOL(\kappa)$, which returns the logical specification of κ. In essence, $toFOL(\kappa)$ specifies the evaluation context and the redex within κ, given in Figure 3. It is important to note that HP constructs, evaluation contexts, and states appear as predicate arguments in this presentation to enhance readability. Their syntax can be expressed as string literals to conform with the syntax of predicate logic.

We define the logical specification of traces for both finite and infinite cases based on the logical specification of configurations, using $toFOL(\kappa)$. Let $\tau = \kappa_0\kappa_1 \cdots \kappa_n$

for some n, then its logical specification is defined as $toFOL(\tau) = \bigcup_{i=0}^{n} toFOL(\kappa_i)$. Otherwise, for infinite trace $\tau = \kappa_0\kappa_1\cdots$, $toFOL(\tau) = \bigcup_{\tau' \in prefix(\tau)} toFOL(\tau')$, where $toFOL(\tau') = \bigcup_{i=0}^{n} toFOL(\kappa_i)$, for $\tau' = \kappa_0\kappa_1\cdots\kappa_n$. Showing that $toFOL(\tau)$ is injective and monotonically increasing can be done straightforwardly.

3.2 Instantiation of Logging Specifications

The class of logging specifications $\mathcal{LS}_{call}$ is defined to specify temporal relations among function invocations in HPs. Formally, $\mathcal{LS}_{call}$ is the set of all logging specifications LS given by $spec(\Gamma_G, \{\text{LoggedCall}\})$, where Γ_G is a set of Horn clauses referred to as *guidelines*, including clauses of the form

$$\begin{aligned} &\forall t_0, \cdots, t_n, \bar{x}_0, \cdots, \bar{x}_n. \\ &\quad \text{Call}(t_0, f_0, \bar{x}_0) \bigwedge_{i=1}^{n} \big(\text{Call}(t_i, f_i, \bar{x}_i) \wedge t_i < t_0\big) \wedge \\ &\quad \varphi(t_0, \cdots, t_n) \wedge \varphi'(\bar{x}_0, \cdots, \bar{x}_n) \implies \text{LoggedCall}(f_0, \bar{x}_0), \end{aligned} \tag{3.1}$$

in which for all $j \in \{0, \cdots, n\}$

- f_j is a function name with a definition in $\mathcal{C}$,
- $\bar{x}_j$ is a placeholder for a sequence of parameters passed to f_j, and
- $\text{Call}(t_j, f_j, \bar{x}_j)$ specifies the event of invoking f_j at time t_j with parameters $\bar{x}_j$.

In (3.1), $\varphi(t_0, \cdots, t_n)$ epresents a potentially empty conjunctive sequence of literals of the form $t_i < t_j$. Additionally, we define *triggers* and *logging events* as follows: $Triggers(LS) = \{f_1, \cdots, f_n\}$ and $Logevent(LS) = f_0$. The *logging preconditions* are predicates $\text{Call}(t_i, f_i, \bar{x}_i)$ for all $i \in \{1, \cdots, n\}$.

Example 3.2. We can define the guideline of the logging specification for the example in Section 1.1 and Example 3.1 as follows:

$$\begin{aligned} &\forall t_0, t_1, t_2, t_3, \bar{x}_0, \cdots, \bar{x}_n. \text{Call}(t_0, evolve, \bar{x}_0) \wedge \\ &\quad \text{Call}(t_1, cmdHB, \bar{x}_2) \wedge t_1 < t_0 \wedge \\ &\quad \text{Call}(t_2, assess, \bar{x}_2) \wedge t_2 < t_0 \wedge \\ &\quad \text{Call}(t_3, sen, \bar{x}_3) \wedge t_3 < t_0 \wedge \\ &\quad t_3 < t_2 \wedge t_2 < t_1 \implies \text{LoggedCall}(evolve, \bar{x}_0) \end{aligned}$$

The guidelines ensures logging the invocations of *evolve*() if the functions *sen*(), *assess*(), and *cmdHB*() are called in order before *evolve*(). In this specification, we are skipping to list a detailed collection of inputs for each function for the sake of brevity. As mentioned in Section 1.1, for instance, $\bar{x}_0$ could include different environmental variables, e.g., the location, speed, and acceleration.

3.3 Target HP Model

We extend $\mathcal{HP}$ to define the target program model, denoted by $\mathcal{HP}_{\log}$, with the following syntax and semantics. The task of the instrumentation algorithm is to map a program specified in $\mathcal{HP}$ to a program in $\mathcal{HP}_{\log}$.

3.3.1 Syntax

HPs are expanded with two additional constructs: $\alpha ::= \cdots \mid \mathsf{callEvent}(f, \bar{e}) \mid \mathsf{emit}(f, \bar{e})$. It is important to note that $\bar{e}$ is treated as a list of expressions rather than a sequence of them in callEvent and emit, ensuring they have fixed arities.

3.3.2 Semantics

We extend the configurations in $\mathcal{HP}_{\log}$ with two additional components, as well. A configuration is defined as $\kappa ::= (t, \omega, \alpha, \Sigma, \Lambda)$, where Σ is a set of predicates of the form $\text{Call}(t, f, \bar{r})$, where $f \in \mathit{Triggers}$. These preconditions are supposed to be gathered, in order to decide whether to log an event. Λ is the audit log, i.e., the set of predicates of the form $\text{LoggedCall}(f, \bar{r})$. The initial configuration is $\kappa_0 = (0, \omega_0, \alpha, \emptyset, \emptyset)$.

Figure 4 depicts the reduction semantics of $\mathcal{HP}_{\log}$. Note that $\mathcal{C}$ and Γ_G are part of runtime structure, but for the sake of breveity we avoid annotating each step of reduction with these static structures.

$\mathcal{HP}_{\log}$ inherits the operational semantics of $\mathcal{HP}$ via rule HP. Rule CALL_EV handles reduction with the $\mathsf{callEvent}(f, \bar{e})$ statement, adding $\text{Call}(t, f, \omega[\![\bar{e}]\!])$ to Σ. For the $\mathsf{emit}(f, \bar{e})$ statement, rule LOG checks if the predicate $\text{LoggedCall}(f, \bar{r})$ is derivable from Σ and Γ_G. If so, it adds it to the audit log Λ; otherwise, there is no change to the log, as specified by rule NO_LOG.

The residual log of a configuration is given by $\mathit{logof}(\kappa) = \mathbb{L} = \Lambda$, where $\kappa = (\_, \_, \_, \_, \Lambda)$. This instantiation defines $\tau \rightsquigarrow \mathbb{L}$ for $\mathcal{HP}_{\log}$ as per Definition 2.4. Since $\mathbb{L}$ consists of logical literals, $\mathit{toFOL}(\mathbb{L}) = \mathbb{L}$ adequately specifies $\lfloor \mathbb{L} \rfloor$ in line with Definition 2.10.

3.4 Instrumentation of HPs

In this section, we discuss the instrumentation algorithm tailored for audit logging in HPs. Next, we review how the semantic preservation can be defined for this algorithm. Lastly, we specify the main results.

3.4.1 Instrumentation algorithm

Instrumentation algorithm $\mathcal{I}_{\mathcal{HP}}$ takes an $\mathcal{HP}$ program $\mathfrak{p} = \langle \alpha, \mathcal{C} \rangle$ and a logging specification $LS \in \mathcal{LS}_{call}$, and produces a program $\mathfrak{p}' = \langle \alpha_{\mathfrak{p}}, \mathcal{C}' \rangle$ in $\mathcal{HP}_{\log}$. The details of how $\mathcal{I}_{\mathcal{HP}}$ modifies the codebase $\mathcal{C}$ is given in Figure 5. Intuitively, with the invocation of a function f:

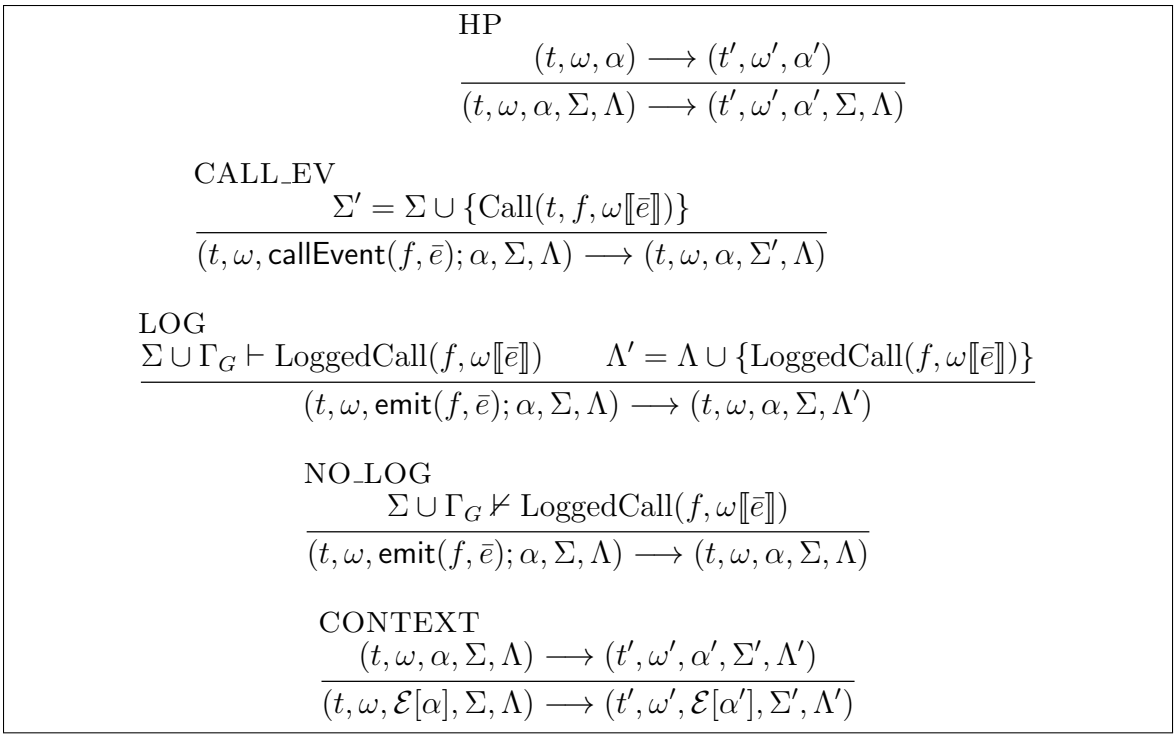

Figure 4: Operational semantics of $\mathcal{HP}_{\log}$.

- If the invocation of f serves as a trigger, its execution must be preceded by a callEvent statement. Consequently, the invocation of f is recorded in the set of logging preconditions Σ, as outlined by the rule CALL_EV.
- If the invocation of f constitutes a logging event, its execution must also be preceded by a callEvent statement, similar to the previous scenario. Subsequently, the system evaluates whether this invocation should be logged, which is determined by the presence of an emit statement (covered by rules LOG and NO_LOG in Figure 4). Following this evaluation, f proceeds with its execution as usual.
- If the invocation of f does not serve as a trigger nor a logging event, then the function executes without any alteration in behavior.

Example 3.3. For the logging specification in Example 3.2, $\mathcal{I}_{\mathcal{HP}}$ injects $\mathsf{callEvent}(sen,\bar{x})$, $\mathsf{callEvent}(assess,\bar{x})$ and $\mathsf{callEvent}(cmdHB,\bar{x})$ to the beginning of functions $sen()$, $assess()$ and $cmdHB()$, respectively, as these functions are triggers to log. Since $evolve()$ is the logging event, $\mathcal{I}_{\mathcal{HP}}$ modifies the body of the function to be

$$\mathsf{callEvent}(evolve,\bar{x});\mathsf{emit}(evolve,\bar{x});(k'=1,x'=v,v'=a \ \&\ k \leq p).$$

$$\text{TRIGGER}\quad \frac{f \in \mathit{Triggers}(LS) \qquad \mathcal{C}(f) = [f(\bar{x}) = e]}{\mathcal{C}'(f) = [f(\bar{x}) = \mathsf{callEvent}(f, \bar{x}); e]}$$

$$\text{LOG\_EV}\quad \frac{f \in \mathit{Logevent}(LS) \qquad \mathcal{C}(f) = [f(\bar{x}) = e]}{\mathcal{C}'(f) = [f(\bar{x}) = \mathsf{callEvent}(f, \bar{x}); \mathsf{emit}(f, \bar{x}); e]}$$

$$\text{NO\_TRIGGER\_LOG\_EV}\quad \frac{f \notin \mathit{Triggers}(LS) \cup \mathit{Logevent}(LS)}{\mathcal{C}'(f) = \mathcal{C}(f)}$$

Figure 5: Definition of $\mathcal{I}_{\mathcal{HP}}(\langle \alpha_{\mathfrak{p}}, \mathcal{C} \rangle) = \langle \alpha_{\mathfrak{p}}, \mathcal{C}' \rangle$.

$$\mathit{trim}(\mathsf{callEvent}(f, \bar{e})) = \epsilon \qquad \mathit{trim}(\mathsf{emit}(f, \bar{e})) = \epsilon$$

$$\mathit{trim}(\alpha_1 \cup \alpha_2) = \mathit{trim}(\alpha_1) \cup \mathit{trim}(\alpha_2) \qquad \mathit{trim}(\alpha_1; \alpha_2) = \mathit{trim}(\alpha_1); \mathit{trim}(\alpha_2)$$

$$\mathit{trim}(\alpha^*) = (\mathit{trim}(\alpha))^* \qquad \mathit{trim}(\alpha) = \alpha \quad \text{(Otherwise)}$$

Figure 6: Function *trim*.

3.4.2 Instantiation of trace correspondence relation $\coloneqq\!\approx$

We instantiate the abstraction of the correspondence relation $\coloneqq\!\approx$ between source and target traces, as defined in Definition 2.5, for $\mathcal{I}_{\mathcal{HP}}$. We establish the source and target trace correspondence relation as follows: $\tau_1\kappa_1 \coloneqq\!\approx \tau_2\kappa_2$ if $\kappa_1 = (t, \omega, \alpha_1)$, $\kappa_2 = (t, \omega, \alpha_2, \Sigma, \Lambda)$, and $\mathit{trim}(\alpha_2) = \alpha_1$. The function *trim*, outlined in Figure 6, essentially eliminates any callEvent and emit statements that $\mathcal{I}_{\mathcal{HP}}$ may introduce to an HP.

3.4.3 Main Results

Main properties include two results. The instrumentation algorithm $\mathcal{I}_{\mathcal{HP}}$ is semantics preserving, and is correct.

Lemma 3.4. $(0, \omega_0, \alpha_{\mathfrak{p}}) \coloneqq\!\approx (0, \omega_0, \alpha_{\mathcal{I}_{\mathcal{HP}}(\mathfrak{p}, LS)}, \emptyset, \emptyset)$.

Proof. It is straightforward to show that $\mathit{trim}(\alpha_{\mathcal{I}_{\mathcal{HP}}(\mathfrak{p}, LS)}) = \alpha_{\mathfrak{p}}$ as $\alpha_{\mathfrak{p}}$ simply invokes $\mathit{main}()$. □

Lemma 3.5. *If* $(t, \omega, \mathcal{E}[\mathit{trim}(\alpha)], \Sigma, \Lambda) \longrightarrow (t', \omega', \mathcal{E}[\alpha'], \Sigma', \Lambda')$*, then there exist* t''*,* ω''*,* α''*,* Σ''*, and* Λ'' *such that* $(t, \omega, \mathcal{E}[\alpha], \Sigma, \Lambda) \longrightarrow^* (t'', \omega'', \mathcal{E}[\alpha''], \Sigma'', \Lambda'')$ *and* $\mathit{trim}(\alpha'') = \mathit{trim}(\alpha')$*.*

Proof. By induction on the structure of α. Nontrivial cases are $\alpha = \mathsf{callEvent}(f, \bar{x}); \beta$ and $\alpha = \mathsf{callEvent}(f, \bar{x}); \mathsf{emit}(f, \bar{x}); \beta$. □

Lemma 3.6. *If* $\tau_1(t_1, \omega_1, \alpha_1) \coloneqq\!\approx \tau_2(t_2, \omega_2, \alpha_2, \Sigma_2, \Lambda_2)$ *and* $(t_1, \omega_1, \alpha_1) \longrightarrow (t_1', \omega_1', \alpha_1')$*, then there exists some* τ_2' *such that* $(t_2, \omega_2, \alpha_2, \Sigma_2, \Lambda_2) \Downarrow \tau_2'$ *and* $\tau_1(t_1, \omega_1, \alpha_1)(t_1', \omega_1', \alpha_1') \coloneqq\!\approx \tau_2\tau_2'$*.*

$$\kappa \Downarrow \kappa \qquad\qquad \frac{\kappa \longrightarrow \kappa' \qquad \kappa' \Downarrow \tau}{\kappa \Downarrow \kappa'\tau}$$

Figure 7: Inductive definition of $\kappa \Downarrow \tau$

Proof. By induction on the derivation of $(t_1, \omega_1, \alpha_1) \longrightarrow (t_1', \omega_1', \alpha_1')$ and applying Lemma 3.5. □

Lemma 3.7. *If $\tau_1(t_1, \omega_1, \alpha_1) :\approx \tau_2(t_2, \omega_2, \alpha_2, \Sigma_2, \Lambda_2)$ and $(t_1, \omega_1, \alpha_1) \Downarrow \tau_1'$, then there exists some τ_2' such that $(t_2, \omega_2, \alpha_2, \Sigma_2, \Lambda_2) \Downarrow \tau_2'$ and $\tau_1\tau_1' :\approx \tau_2\tau_2'$.*

Proof. By induction on the derivation of $(t_1, \omega_1, \alpha_1) \Downarrow \tau_1'$ and applying Lemma 3.6. Note that $\kappa \Downarrow \tau$ can be inductive defined as given in Figure 7. □

Lemma 3.8. *If $\tau_1(t_1, \omega_1, \alpha_1) :\approx \tau_2\kappa_2$ and $\kappa_2 \longrightarrow \kappa_2'$, then there exists some τ_1' such that $(t_1, \omega_1, \alpha_1) \Downarrow \tau_1'$ and $\tau_1\tau_1' :\approx \tau_2\kappa_2\kappa_2'$.*

Proof. By induction on the derivation of $\kappa_2 \longrightarrow \kappa_2'$. □

Lemma 3.9. *If $\tau_1(t_1, \omega_1, \alpha_1) :\approx \tau_2\kappa_2$ and $\kappa_2 \Downarrow \tau_2'$, then there exists some τ_1' such that $(t_1, \omega_1, \alpha_1) \Downarrow \tau_1'$ and $\tau_1\tau_1' :\approx \tau_2\tau_2'$.*

Proof. By induction on the derivation of $\kappa_2 \Downarrow \tau_2'$ and applying Lemma 3.8. □

Theorem 3.10 (Semantics preservation)**.** *$\mathcal{I}_{\mathcal{HP}}$ is semantics preserving (Definition 2.5).*

Proof. Lemmas 3.4, 3.7 and 3.9 entail the result. Note that Lemmas 3.4 and 3.7 satisfy the first condition of Definition 2.5 and Lemmas 3.4 and 3.9 provide the satisfaction of the second condition of Definition 2.5. □

Lemma 3.11. *The following propositions hold for closure in FOL and least Herbrand model ($\mathcal{H}$).*

- $\mathit{Closure}(\mathcal{H}(\Gamma)) = \mathit{Closure}(\Gamma)$
- $\mathcal{H}(\Gamma) = \mathcal{H}(\mathit{Closure}(\Gamma))$
- $\mathit{Closure}(\mathit{Closure}(\mathcal{H}(\Gamma) \cap \Gamma')) = \mathit{Closure}(\mathcal{H}(\Gamma) \cap \Gamma')$

Lemma 3.12. *If $(t, \omega, \alpha, \Sigma, \Lambda) \longrightarrow (t', \omega', \alpha', \Sigma', \Lambda')$ and $\mathrm{LoggedCall}(f, \bar{x}) \in \Lambda' - \Lambda$, then there exist $\mathcal{E}$ and β such that $\alpha = \mathcal{E}[\mathsf{emit}(f, \bar{x}); \beta]$.*

Proof. By induction on the derivation of $(t, \omega, \alpha, \Sigma, \Lambda) \longrightarrow (t', \omega', \alpha', \Sigma', \Lambda')$. □

Lemma 3.13. *If $(t, \omega, \alpha, \Sigma, \Lambda) \Downarrow \tau(t', \omega', \alpha', \Sigma', \Lambda')$ and $\mathrm{LoggedCall}(f, \bar{x}) \in \Lambda' - \Lambda$, then there exist $\mathcal{E}$, β, τ', κ_1, κ_2, and α_1 such that $\alpha_1 = \mathcal{E}[\mathsf{emit}(f, \bar{x}); \beta]$, $\kappa_1 = (\_, \_, \alpha_1, \_, \_)$ and $\tau'\kappa_1\kappa_2 \in \mathit{prefix}(\tau(t', \omega', \alpha', \Sigma', \Lambda'))$.*

Proof. By induction on the derivation of $t, \omega, \alpha, \Sigma, \Lambda) \Downarrow \tau(t', \omega', \alpha', \Sigma', \Lambda')$ and applying Lemma 3.12. □

Let $[\cdots t]\tau$ denote the prefix of τ of length t.

Lemma 3.14. *If $\mathcal{I}(\mathfrak{p}, LS) \Downarrow \tau'$, $\tau :\approx \tau'$, and $\tau' \rightsquigarrow \mathbb{L}$, then $toFOL(\mathbb{L}) = \mathcal{H}(\Gamma_G \cup toFOL(\tau)) \cap FOL(\{\text{LoggedCall}\})$.*

Proof. We first show that $toFOL(\mathbb{L}) \subseteq \mathcal{H}(\Gamma_G \cup toFOL(\tau)) \cap FOL(\{\text{LoggedCall}\})$. Let $tail(\tau') = (\_, \_, \_, \_, \Lambda)$. Assume that $\text{LoggedCall}(f, \bar{x}) \in toFOL(\mathbb{L})$. Then, according to Lemma 3.13, there exist some trace $\hat{\tau}$ and configurations κ_1 and κ_2 such that $\hat{\tau}\kappa_1\kappa_2 \in prefix(\tau')$, $\kappa_1 = (\_, \_, \alpha_1, \_, \_)$, and $\alpha_1 = \mathcal{E}[\mathsf{emit}(f, \bar{x}); \beta]$. By Theorem 3.10, we know that there exist some source trace τ_0 such that $\mathfrak{p} \Downarrow \tau_0$, and it simulates the target trace $\hat{\tau}\kappa_1\kappa_2$, i.e., $\tau_0 :\approx \hat{\tau}\kappa_1\kappa_2$. According to $\mathcal{I}_{\mathcal{HP}}$ definition, statement emit can only appear in the body of a log event function. Therefore, preconditions of rule (3.1) are satisfied by $\Gamma_G \cup toFOL(\tau_0)$. Due to $\tau_0 \in prefix(\tau)$ and monotonicity of $toFOL()$, the precondition of rule (3.1) are satisfied by $\Gamma_G \cup toFOL(\tau)$. Therefore, $\text{LoggedCall}(f, \bar{x}) \in \mathcal{H}(\Gamma_G \cup toFOL(\tau)) \cap FOL(\{\text{LoggedCall}\})$.

Next, we show the following: $toFOL(\mathbb{L}) \supseteq \mathcal{H}(\Gamma_G \cup toFOL(\tau)) \cap FOL(\{\text{LoggedCall}\})$. Assume that $\text{LoggedCall}(f, \bar{x}) \in \mathcal{H}(\Gamma_G \cup toFOL(\tau)) \cap FOL(\{\text{LoggedCall}\})$. This entails $\text{Call}(t, f, \bar{x}) \in toFOL(\tau)$ for some t in which preconditions of rule (3.1) are satisfied. Then, $tail([\cdots t]\tau) = \mathcal{E}[f(\bar{x})]$ for some $\mathcal{E}$. According to Theorem 3.10, there exists some target trace $\hat{\tau}$ such that $\mathcal{I}_{\mathcal{HP}}(\mathfrak{p}, LS) \Downarrow \hat{\tau}$, and $[\cdots t]\tau :\approx \hat{\tau}$. Let $tail(\hat{\tau}) = (\_, \_, \hat{\alpha}, \_, \_)$. Then, $trim(\hat{\alpha}) = \mathcal{E}[f(\bar{x})]$. We have

$$(\hat{t}, \hat{\omega}, \mathcal{E}[trim(\hat{\alpha})], \hat{\Sigma}, \hat{\Lambda}) \longrightarrow (\hat{t}, \hat{\omega}, \mathcal{E}[\mathsf{callEvent}(f, \bar{x}); \mathsf{emit}(f, \bar{x}); \beta], \hat{\Sigma}, \hat{\Lambda})$$

By Lemma 3.5, we then have some trace τ_0, and configurations $\kappa_1, \cdots, \kappa_4$ such that $(\hat{t}, \hat{\omega}, \mathcal{E}[\hat{\alpha}], \hat{\Sigma}, \hat{\Lambda}) \Downarrow \tau_0\kappa_0\kappa_1 \cdots \kappa_4$, where

- $\kappa_1 = (\_, \_, \mathcal{E}'[f(\bar{x})], \_, \hat{\Lambda})$,
- $\kappa_2 = (\_, \_, \mathcal{E}'[\mathsf{callEvent}(f, \bar{x}); \mathsf{emit}(f, \bar{x}); \beta], \_, \hat{\Lambda})$,
- $\kappa_3 = (\_, \_, \mathcal{E}'[\mathsf{emit}(f, \bar{x}); \beta], \_, \hat{\Lambda})$, and
- $\kappa_4 = (\_, \_, \mathcal{E}'[\beta], \_, \hat{\Lambda}')$.

Note that $\text{LoggedCall}(f, \bar{x}) \in \hat{\Lambda}'$, which entails that if $\hat{\tau}\tau_0\kappa_1 \cdots \kappa_4 \rightsquigarrow \hat{\mathbb{L}}$ then $\text{LoggedCall}(f, \bar{x}) \in \hat{\mathbb{L}}$. Since $[\cdots (t+1)]\tau :\approx \hat{\tau}\tau_0\kappa_1 \cdots \kappa_4$ and $[\cdots (t+1)]\tau \in prefix(\tau)$, for any trace τ' where $\tau :\approx \tau'$, if $\tau' \rightsquigarrow \mathbb{L}$, then $\hat{\mathbb{L}} \subseteq \mathbb{L}$ due to monotonicity of log growth. This entails that $\text{LoggedCall}(f, \bar{x}) \in \mathbb{L}$, and thus $\text{LoggedCall}(f, \bar{x}) \in toFOL(\mathbb{L})$. □

Theorem 3.15 (Instrumentation correctness)**.** *$\mathcal{I}_{\mathcal{HP}}$ is correct (Definition 2.6).*

Proof. Let $\mathfrak{p}$ be a source system and LS be a logging specification defined as Section 3.2. If $\mathcal{I}(\mathfrak{p}, LS) \Downarrow \tau'$, $\tau :\approx \tau'$, and $\tau' \rightsquigarrow \mathbb{L}$, then we need to show that $\mathit{Closure}(\mathit{toFOL}(\mathbb{L})) = LS(\tau)$. By Lemma 3.14, we have $\mathit{toFOL}(\mathbb{L}) = \mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$. By Lemma 3.11, we have

$$\begin{aligned} LS(\tau) &= \mathit{Closure}(\mathit{Closure}(\mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau))) \cap \mathit{FOL}(\{\text{LoggedCall}\})) \\ &= \mathit{Closure}(\mathcal{H}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})) \end{aligned}$$

This entails the result. □

4 Related Work

Hybrid-dynamic models may use programming languages techniques to specify CPSs, particularly through HPs, as we have employed in this report. Along with HPs, there are two other major approaches to formally model CPSs: Hybrid automata, which describe a hybrid system through a finite state transition system that captures both discrete and continuous variables in each state [3, 15], and hybrid process calculi [14, 19, 20, 21], which model CPSs in terms of agents representing physical plants and cyber components. These agents communicate through named channels, and in systems like CCPS [20], a labeled transition system demonstrates how such agents execute using shared channels.

There is a rich body of work on audit logging in CPSs as one of the necessary components of cybersecurity assurance in this domain [23]. More recent examples include developing an accountability system for autonomous robots [13], a microservices-based architecture for industrial Internet of Things and CPSs [12], and enhancing realtime CPSs with audit logging capabilities [10, 11]. These proposals have architectural approaches to audit logging, whereas we study audit logging from a programming point of view and establish formal guarantees about the quality of the generated logs in CPSs.

Information-algebraic models [17] have been used within the last decade to specify and enforce correct audit logging in different realms of computation, initially in linear functional settings [5]. Moreover, this framework has been utilized to identify and analyze direct information flows in Java-like languages [7, 27]. It has also inspired investigations into audit logging correctness in concurrent systems [6], which has facilitated the study of correct audit logging in microservices [8, 1].

5 Conclusion and Future Work

In this technical report, we present an implementation model aimed at ensuring accurate audit logging in hybrid-dynamic systems. Our approach involves an algorithm that instruments existing hybrid programs based on a formal specification of audit logging requirements. In order to specify the audit logging requirements Horn clause

logic is utilized. These specifications establish temporal relationships among runtime events. We have demonstrated that our algorithm preserves the semantics of the original program, except for operations related to audit logging. Furthermore, we have formally validated that our approach reliably produces correct audit logs, thereby preventing both the omission of necessary logging events and the inclusion of superfluous ones.

In future work, we plan to extend the scope of our current findings by applying them to practical, real-world programming environments for CPSs. This progression will involve adapting the foundational results obtained in this study to different languages and technologies that are widely used in CPSs. By doing so, we aim to test the robustness and applicability of our results under different and possibly more complex conditions, potentially leading to further refinements and enhancements of our audit logging methodology. This approach will help in optimizing them for practical deployment in actual systems, thereby bridging the gap between the current research and real-world application.

References

[1] Ahn, N.D., Amir-Mohammadian, S.: Instrumenting microservices for concurrent audit logging: Beyond Horn clauses. In: Proceedings of the 16th IEEE International Workshop on Security, Trust & Privacy for Software Applications (STPSA 2022), as part of the 46th Annual IEEE Computers, Software, and Applications Conference (COMPSAC 2022). pp. 1762–1767 (June 2022)

[2] Alladi, T., Chamola, V., Zeadally, S.: Industrial control systems: Cyberattack trends and countermeasures. Computer Communications 155, 1–8 (2020)

[3] Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Hybrid systems, pp. 209–229. Springer (1992)

[4] Amir-Mohammadian, S.: A Formal Approach to Combining Prospective and Retrospective Security. Ph.D. thesis, The University of Vermont (July 2017)

[5] Amir-Mohammadian, S., Chong, S., Skalka, C.: Correct audit logging: Theory and practice. In: Principals of Security and Trust. pp. 139–162 (2016)

[6] Amir-Mohammadian, S., Kari, C.: Correct audit logging in concurrent systems. Electronic Notes in Theoretical Computer Science 351, 115–141 (September 2020)

[7] Amir-Mohammadian, S., Skalka, C.: In-depth enforcement of dynamic integrity taint analysis. In: Programming Languages and Analysis for Security (2016)

[8] Amir-Mohammadian, S., Zowj, A.Y.: Towards concurrent audit logging in microservices. In: Proceedings of the 15th IEEE International Workshop on Security, Trust & Privacy for Software Applications (STPSA 2021), as part of the 45th Annual IEEE Computers, Software, and Applications Conference (COMPSAC 2021). pp. 1357–1362 (July 2021)

[9] Ávila, R., Khoury, R., Khoury, R., Petrillo, F.: Use of security logs for data leak detection: a systematic literature review. Security and communication networks 2021, 1–29 (2021)

[10] Bansal, A., Kandikuppa, A., Chen, C.Y., Hasan, M., Bates, A., Mohan, S.: Towards efficient auditing for real-time systems. In: European Symposium on Research in Computer Security. pp. 614–634. Springer (2022)

[11] Bansal, A., Kandikuppa, A., Hasan, M., Chen, C.Y., Bates, A., Mohan, S.: System auditing for real-time systems. ACM Transactions on Privacy and Security 26(4), 1–37 (2023)

[12] Dobaj, J., Iber, J., Krisper, M., Kreiner, C.: A microservice architecture for the industrial internet-of-things. In: Proceedings of the 23rd European Conference on Pattern Languages of Programs. pp. 1–15 (2018)

[13] Fernández-Becerra, L., Manuel Guerrero-Higueras, Á., Rodríguez-Lera, F.J., Matellán, V.: Accountability as a service for robotics: Performance assessment of different accountability strategies for autonomous robots. Logic Journal of the IGPL 32(2), 243–262 (2024)

[14] Galpin, V., Bortolussi, L., Hillston, J.: Hype: Hybrid modelling by composition of flows. Formal Aspects of Computing 25(4), 503–541 (2013)

[15] Henzinger, T.A.: The theory of hybrid automata. In: Verification of digital and hybrid systems, pp. 265–292. Springer (2000)

[16] Kim, K., Kim, J.S., Jeong, S., Park, J.H., Kim, H.K.: Cybersecurity for autonomous vehicles: Review of attacks and defense. Computers & security 103, 102150 (2021)

[17] Kohlas, J., Schmid, J.: An algebraic theory of information: An introduction and survey. Information 5(2), 219–254 (2014)

[18] Kumar, P., Lin, Y., Bai, G., Paverd, A., Dong, J.S., Martin, A.: Smart grid metering networks: A survey on security, privacy and open research issues. IEEE Communications Surveys & Tutorials 21(3), 2886–2927 (2019)

[19] Lanese, I., Bedogni, L., Di Felice, M.: Internet of things: a process calculus approach. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. pp. 1339–1346 (2013)

[20] Lanotte, R., Merro, M.: A calculus of cyber-physical systems. In: International Conference on Language and Automata Theory and Applications. pp. 115–127. Springer (2017)

[21] Lanotte, R., Merro, M.: A semantic theory of the internet of things. Information and Computation 259, 72–101 (2018)

[22] Lehto, M.: Cyber-attacks against critical infrastructure. In: Cyber security: Critical infrastructure protection, pp. 3–42. Springer (2022)

[23] Mitchell, R., Chen, I.R.: A survey of intrusion detection techniques for cyber-physical systems. ACM Computing Surveys (CSUR) 46(4), 1–29 (2014)

[24] Platzer, A.: Differential dynamic logic for hybrid systems. Journal of Automated Reasoning 41(2), 143–189 (2008)

[25] Platzer, A.: The complete proof theory of hybrid systems. In: 2012 27th Annual IEEE Symposium on Logic in Computer Science. pp. 541–550. IEEE (2012)

[26] Platzer, A.: Logical foundations of cyber-physical systems, vol. 662. Springer (2018)

[27] Skalka, C., Amir-Mohammadian, S., Clark, S.: Maybe tainted data: Theory and a case study. Journal of Computer Security 28(3), 295–335 (April 2020)