Technical Report: Correct Audit Logging in Concurrent Systems with Negative Triggers

Sepehr Amir-Mohammadian

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

Abstract

In previous work [3], we had proposed the semantics of correct audit logging in concurrent systems using information algebra, and then had proposed an implementation model in π-calculus that instruments programs to generate correct audit logs according to the proposed semantic framework. The proposed instrumentation algorithm receives audit logging requirements specified in Horn clause logic that specify a collection of events (positive triggers) as preconditions to log. In this work, we extend the formal specification of audit logging requirements with negative triggers (along with positive ones) to boost the expressivity of the audit logging policies, i.e., a collection of events that should not happen as precondition to log. This way, our language of audit logging requirement specifications goes beyond Horn clauses. We show that the original instrumentation algorithm (from [3]) is potent enough to emit programs that generate correct audit logs at runtime.

Keywords— Audit logging, Concurrent systems, Programming languages, Security

1 Semantics of Audit Logging

In order to provide a standalone formal presentation, in this section we review the information-algebraic semantics of audit logging and the instantiation of the semantic framework with first-order logic, which is originally proposed by Amir-Mohammadian et al. [2]. We have applied minor modifications to the model to better suit concurrency and nondeterministic runtime behavior, inherent to concurrent systems.

1.1 Information-Algebraic Semantic Framework

In order to specify how audit logs are generated at runtime, we need to abstract system states and their evolution through the computation. A *system configuration*

κ abstracts the state of the system at a given point during the execution. Let $\mathcal{K}$ denote the set of system configurations. We posit a binary reduction relation among configurations, i.e., $(\longrightarrow) \subseteq \mathcal{K} \times \mathcal{K}$ which denotes the computational steps, and is used in the standard infix form.[1] A *system trace* τ is a potentially infinite sequence of system configurations, i.e., $\tau = \kappa_0\kappa_1 \cdots$, where κ_i is the ith configuration in sequence, and $\kappa_i \longrightarrow \kappa_{i+1}$. We denote the set of all traces by $\mathcal{T}$, and define *prefix*(τ) as the set of all prefixes of τ.

Information algebra is used to define the notion of correctness for audit logs. In Section 1.2, we instantiate this abstract algebraic structure to model a specific class of audit logging requirements. We define an information algebra in the following.

Definition 1.1 (Information algebra)**.** An information algebra (Φ, Ψ) is a two-sorted algebra consisting of an Abelian semigroup of information elements, Φ, as well as a lattice of querying domains, Ψ. Two fundamental operators are presumed in this algebra: a combination operator, $(\otimes) : \Phi \times \Phi \to \Phi$, and a focusing operator, $(^{\Rightarrow}) : \Phi \times \Psi \to \Phi$. An Information algebra (Φ, Ψ) satisfies a set of properties, in connection to combination and focusing operators.[2] We let $X, Y, Z, \cdots$ to range over elements of Φ, and E range over Ψ.

$X, Y \in \Phi$ are information elements that can be combined to make a more inclusive information element $X \otimes Y$. $E \in \Psi$ is a querying domain with a certain level of granularity that is used by the focusing operator to extract information from an information element X, denoted by $X^{\Rightarrow E}$. For example, relational algebra is an instance of information algebra, in which relations instantiate information elements, sets of attributes instantiate querying domains, natural join of two relations defines the combination operator, and projection of a relation on a set of attributes defines the focusing operator [6].

Combination of information elements induces a partial order relation $(\preccurlyeq) \subseteq \Phi \times \Phi$ among information elements, defined as follows: $X \preccurlyeq Y$ iff $X \otimes Y = Y$. Intuitively, $X \preccurlyeq Y$ means that Y contains the information element X.

As part of the semantics of audit logging, we treat execution traces as information elements, i.e., the information content of the execution trace. To this end, we posit $\lfloor\cdot\rfloor : \mathcal{T} \to \Phi$ as a mapping in which, intuitively, $\lfloor\tau\rfloor$ refers to the information content of the trace τ. We also impose the condition that $\lfloor\cdot\rfloor$ be injective and monotonically increasing, i.e., if $\tau' \in$ *prefix*(τ) then $\lfloor\tau'\rfloor \preccurlyeq \lfloor\tau\rfloor$. This ensure that as the execution trace grows in length, it contains more information.

In the following definition, we define audit logging requirements in an abstract form. We call this abstraction a *logging specification.* This definition is abstract enough to encompass different execution models, as well as different representations

[1] A notational convention throughout the paper is that infix operators and relations are wrapped with parentheses when their signature are specified.

[2] We avoid discussing these properties in detail here for the sake of brevity. Readers are referred to [6] for the complete formulation.

of information. In Sections 1.2 and 2.2, we instantiate this definition with a more concrete structure that guides us on how to implement audit logging requirements.

Definition 1.2 (Logging specifications)**.** Logging specification LS is defined as a mapping from system traces to information elements, i.e., $LS : \mathcal{T} \to \Phi$. Intuitively, $LS(\tau)$ declares what information must be logged, if the system follows the execution trace τ.

Note that even though $\lfloor \cdot \rfloor$ and LS have the same signature, i.e., maps from traces to information elements, they are conceptually different. $\lfloor \tau \rfloor$ is the whole information contained in τ, whereas $LS(\tau)$ is the information that is supposed to be recorded in the log, if the system follows the execution trace τ.

We denote an audit log with $\mathbb{L}$ which represents a set of data, gathered at runtime. Let $\mathcal{L}$ denote the set of audit logs. In order to judge about the correctness of an audit log, the information content of the audit log needs to be studied in comparison to the information content of the trace that generates that audit log. To this end, we define a mapping that returns the information content of an audit log. We abuse the notation and consider $\lfloor \cdot \rfloor : \mathcal{L} \to \Phi$ as such mapping. Therefore, $\lfloor \mathbb{L} \rfloor$ refers to the information content of the audit log $\mathbb{L}$. We assume that $\lfloor \cdot \rfloor$ on audit logs is injective and monotonically increasing, i.e., if $\mathbb{L} \subseteq \mathbb{L}'$ then $\lfloor \mathbb{L} \rfloor \preccurlyeq \lfloor \mathbb{L}' \rfloor$. Therefore, the more inclusive the audit log is, it contains more information.

The notion of correct audit logging can be defined based on an execution trace and a logging specification. To this end, the information content of the audit log is compared to the information that the logging specification dictates to be recorded in the log, given the execution trace. The following definition captures this relation.

Definition 1.3 (Correctness of audit logs)**.** Audit log $\mathbb{L}$ is *correct* wrt a logging specification LS and a system trace τ iff both $\lfloor \mathbb{L} \rfloor \preccurlyeq LS(\tau)$ and $LS(\tau) \preccurlyeq \lfloor \mathbb{L} \rfloor$ hold. The former refers to the necessity of the information in the audit log, and the latter refers to the sufficiency of those information.

A system that generates audit logs at runtime includes the stored logs as part of its configuration. Let the mapping $\mathit{logof} : \mathcal{K} \to \mathcal{L}$ denote the residual log of a given system configuration, i.e., $\mathit{logof}(\kappa)$ is the set of all recorded audit logs in configuration κ. It is natural to assume that the residual log within configurations grows larger as the execution proceeds. The residual log of a trace is then defined using logof.

Definition 1.4 (Residual log of a system trace)**.** The residual log of a finite system 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 $\tau \rightsquigarrow \mathbb{L}$, then $\mathbb{L}$ is not necessarily correct wrt a given LS and a trace τ. If the residual log of a trace is correct throughout the execution, then that trace is called *ideally-instrumented*. System trace τ is ideally instrumented for a logging specification LS iff for any trace τ' and audit log $\mathbb{L}$, if $\tau' \in \mathit{prefix}(\tau)$ and $\tau' \rightsquigarrow \mathbb{L}$ then $\mathbb{L}$ is correct wrt τ' and LS. Indeed audit logging is an enforceable security property on a trace of execution [2]. Given a logging specification, ideally-instrumented traces induce a safety property [9], and hence implementable by inlined reference monitors [4], and edit automata [1].

Let $\mathfrak{s}$ be a concurrent system with an operational semantics. $\mathfrak{s} \Downarrow \tau$ iff $\mathfrak{s}$ can produce trace τ', either deterministically or non-deterministically, and $\tau \in \mathit{prefix}(\tau')$. We abuse the notation and use $\kappa \Downarrow \tau$ to denote the same concept for configuration κ. We follow program instrumentation techniques, in order to enforce a logging specification on a system. An *instrumentation algorithm* receives the concurrent system as input along with the logging specification, and instruments the system with audit logging capabilities so that the instrumented system generates the required "appropriate" log. An instrumentation algorithm is a partial function $\mathcal{I} : (\mathfrak{s}, LS) \mapsto \mathfrak{s}'$ that instruments $\mathfrak{s}$ according to LS aiming to generate audit logs appropriate for LS. We call $\mathfrak{s}$ the *source system*, and the instrumented system, i.e., $\mathcal{I}(\mathfrak{s}, LS) = \mathfrak{s}'$, the *target system*. Source and target traces refer to the traces of the source and target systems, resp.

It is natural to expect that the instrumentation algorithm would not modify the semantics of the original system drastically. The target system must behave roughly similar to the source system, except for the operations related to audit logging. We call this attribute of an instrumentation algorithm *semantics preservation*, and define it in the following. This definition is abstract enough to encompass different source and target systems (with different runtime semantics), and instrumentation techniques. The abstraction relies on a binary relation $:\approx$, called *correspondence relation*, that relates the source and target traces. Based on different implementations of the source and target systems, and the instrumentation algorithm, the correspondence relation can be defined accordingly.

Definition 1.5 (Semantics preservation by the instrumentation algorithm)**.** Instrumentation algorithm $\mathcal{I}$ is semantics preserving iff for all systems $\mathfrak{s}$ and logging specifications LS, where $\mathcal{I}(\mathfrak{s}, LS)$ is defined, the following conditions hold: 1) For any trace τ, if $\mathfrak{s} \Downarrow \tau$, then there exists some trace τ' such that $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, and $\tau :\approx \tau'$. 2) For any trace τ, if $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau$, then there exists some trace τ' such that $\mathfrak{s} \Downarrow \tau'$, and $\tau' :\approx \tau$.

Another property of the instrumentation algorithm is to ensure that it is *deadlock-free*, meaning that instrumenting a system does not introduce new states being stuck. Let source system $\mathfrak{s}$ generate trace τ, and $\mathcal{I}(\mathfrak{s}, LS)$ generate trace τ' such that $\tau :\approx \tau'$. Then, we call $\mathcal{I}(\mathfrak{s}, LS)$ being stuck if $\mathfrak{s}$ can continue execution following τ (at least for one extra step), while $\mathcal{I}(\mathfrak{s}, LS)$ cannot continue execution following τ'.

Definition 1.6 (Deadlock-freeness of the instrumentation algorithm)**.** Instrumentation algorithm $\mathcal{I}$ is deadlock-free iff for any source system $\mathfrak{s}$, logging specification LS, traces τ and τ', and configuration κ, if $\mathfrak{s} \Downarrow \tau$, $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, $\tau :\approx \tau'$, and $\mathfrak{s} \Downarrow \tau\kappa$, then there exists some configuration κ' such that $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'\kappa'$.

Besides these properties, another important feature of an instrumentation algorithm is the quality of audit logs generated by the instrumented system. The information-algebraic semantic framework provides a platform to define correct instrumentation algorithms for audit logging purposes. Let $\mathfrak{s}$ be a target system, and τ be a source trace. Simulated logs of τ by $\mathfrak{s}$ is the set $\mathit{simlogs}(\mathfrak{s}, \tau)$ defined as $\mathit{simlogs}(\mathfrak{s}, \tau) = \{\mathbb{L} \mid \exists \tau'. \mathfrak{s} \Downarrow \tau' \wedge \tau :\approx \tau' \wedge \tau' \rightsquigarrow \mathbb{L}\}$. Using this set, we can define correctness of instrumentation algorithms in a straightforward manner. Intuitively, the instrumentation algorithm $\mathcal{I}$ is correct if the instrumented system generates audit logs that are correct wrt the logging specification and the source trace. This must hold for any source system, any logging specification, and any possible log generated by the instrumented system.

Definition 1.7 (Correctness of the instrumentation algorithm)**.** Instrumentation algorithm $\mathcal{I}$ is correct iff for all source systems $\mathfrak{s}$, traces τ, and logging specifications LS, $\mathfrak{s} \Downarrow \tau$ implies that for any $\mathbb{L} \in \mathit{simlogs}(\mathcal{I}(\mathfrak{s}, LS), \tau)$, $\mathbb{L}$ is correct wrt LS and τ.

1.2 Instantiation of Logging Specification

In Definition 1.2, logging specification is defined abstractly as a mapping from system traces to information elements. For a more concrete setting, this definition needs to be instantiated with appropriate structures in a way that is useful in the deployment of audit logging. In essence, we need to instantiate information algebra (Definition 1.1). We are interested in logical specification of audit logging requirements due to its easiness of use, expressivity power, well-understood semantics, and off-the-self logic programming engines for subsets of first-order logic (FOL), e.g., Horn clause logic. To this end, in this section, we instantiate information algebra with FOL, which is expressive enough to specify computational events, and the temporal relation among them. Indeed, other variants of logic may also be considered for this purpose.

In order to instantiate information algebra, it is required to 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 1.8, 1.9, and 1.10 accomplish these instantiations.

Definition 1.8 instantiates an FOL-based set of information elements. An information element in our instantiation is a closed set of FOL formulas, under a proof-theoretic deductive system.

Definition 1.8 (Set of closed sets of FOL formulas)**.** Let φ range over FOL formulas, and Γ range over sets of FOL formulas. $\Gamma \vdash \varphi$ denotes a judgment derived by a sound and complete natural deduction proof theory of FOL. We define closure operation *Closure* as $\mathit{Closure}(\Gamma) = \{\varphi \mid \Gamma \vdash \varphi\}$. Then, the set of closed set of FOL formulas is defined as $\Phi_{FOL} = \{\Gamma \mid \Gamma = \mathit{Closure}(\Gamma)\}$.

Definition 1.9 instantiates the lattice of querying domains for the FOL-based information algebra. A query domain is a subset of FOL, defined over certain predicate symbols.

Definition 1.9 (Lattice of FOL sublanguages)**.** Let *Preds* be the set of all assumed predicate symbols along with their arities. If $S \subseteq \mathit{Preds}$, then we denote the sublanguage $\mathit{FOL}(S)$ as the set of well-formed FOL formulas over predicate symbols in S. The set of all such sublanguages $\Psi_{\mathit{FOL}} = \{\mathit{FOL}(S) \mid S \subseteq \mathit{Preds}\}$ is a lattice induced by set containment relation.

Lastly, Definition 1.10 instantiates the combination and focusing operators for the FOL-based information algebra. Combination is the closure of the union of two sets of formulas. Focusing is the closure of the intersection of an information element and a query domain.

Definition 1.10 (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 $\Gamma^{\Rightarrow \mathit{FOL}(S)} = \mathit{Closure}(\Gamma \cap \mathit{FOL}(S))$.

$(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$ is an information algebra, given the Definitions 1.8, 1.9, and 1.10.[3] In order to use $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$ as a framework for audit logging, we also need to instantiate the mapping $\lfloor \cdot \rfloor$, introduced in Section 1.1, to interpret both execution traces and audit logs as information elements.

Definition 1.11 (Mapping traces and audit logs to information elements in $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$)**.** Let $\mathit{toFOL}(\cdot) : (\mathcal{T} \cup \mathcal{L}) \to \mathit{FOL}(\mathit{Preds})$ be an injective and monotonically increasing function. Then, we instantiate $\lfloor \cdot \rfloor = \mathit{Closure}(\mathit{toFOL}(\cdot))$ in order to interpret both traces and logs as information elements in $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$.

Now we can instantiate logging specification LS in the information algebra $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$. To this end, a set of audit logging rules and definitions are assumed to be given in FOL. Let Γ be this set. Moreover, a set of predicate symbols are assumed that reflect on the predicates whose derivation need to be logged at runtime. This set is denoted by S. A logging specification in this setting, receives a trace τ, combines the information content of τ with 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 given rules in Γ and the events in τ.

Definition 1.12 (Logging Specification in $(\Phi_{\mathit{FOL}}, \Psi_{\mathit{FOL}})$)**.** Given 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 $\mathit{spec}(\Gamma, S) = \tau \mapsto (\lfloor \tau \rfloor \otimes \mathit{Closure}(\Gamma))^{\Rightarrow \mathit{FOL}(S)}$.

2 Implementation Model on Concurrent Systems

In this section, we propose an implementation model for correct audit logging in concurrent systems. To this end, we use a variant of π-calculus to specify the concurrent

[3] The reader is referred to [1] for the detailed proof.

system, and propose an instrumentation algorithm that retrofits the system according to a given logging specification. We then specify and prove the properties of interest, including the correctness of the instrumentation algorithm (Definition 1.7).

In Section 2.1, the syntax and semantics of the source system model is introduced. Section 2.2 proposes a class of logging specifications that can specify temporal relations among computational events in concurrent systems. Section 2.3 describes the syntax and semantics of the systems enhanced with audit logging capabilities. Lastly, in Section 2.4, we discuss the instrumentation algorithm and the properties it satisfies.

2.1 Source System Model

We consider a core π-calculus as our source concurrent system model, denoted by Π. One major distinguishing feature of π-calculus is modeling mobile processes using the same category of names for both links and transferable objects, along with scope extrusion. However, mobility is not used in our implementation model. Therefore other seminal process calculi e.g., CSP [5] and CCS [7] can also be considered for this purpose. We employ π-calculus due to its concise syntax and simple semantics that provides a clean and sufficiently abstract specification of the required interactions among concurrent components of the system. The syntax and semantics of the source system are defined in the following. It is based on the representation of the calculus given in [8] which deviates from standard π-calculus by dropping silent prefixes, unguarded summations and labeled reduction system, for the sake of simplicity and conciseness.

2.1.1 Syntax

Let $\mathcal{N}$ be the infinite denumerable set of names, and $a, b, c, \cdots$ and $x, y, z, \cdots$ range over them.

Prefixes Prefixes α are defined as $\alpha ::= a(x) \mid \bar{a}x$. Prefix $a(x)$ is the input prefix, used to receive some name with placeholder x on link a. Prefix $\bar{a}x$ is the output prefix, used to output name x on link a.

Agents and sub-agents Let A, B, C, D range over agent and sub-agent names, and $\mathcal{A}$ be the finite set of such names. Agents (processes) and sub-agents (subprocesses) P are defined as: $P ::= \mathbf{0} \mid \alpha.P \mid (P|P) \mid (\nu x)P \mid C(y_1, \cdots, y_n)$. $\mathbf{0}$ refers to the nil process. $\alpha.P$ provides a sequence of operations in the process; first input/output prefix α takes place, and then P executes. $P|P$ provides parallelism in the system. $(\nu x)P$ restricts (binds) name x within P. $C(y_1, \cdots, y_n)$ refers to the (sub-)agent invocation C with parameters $y_1, \cdots, y_n$. Let P, Q, R range over processes and subprocesses.

Free and bound names Name restriction and input prefix bind names in a process. We denote the set of free names in process P with $fn(P)$. α-conversion for bound names is defined in the standard way.

Notational conventions A sequence of names is denoted by $\tilde{a}$, i.e., $a_1, \cdots, a_l$ for some l. A sequence of name restrictions in a process $(\nu a_1)(\nu a_2)\cdots(\nu a_l)P$ is shown by $(\nu a_1 a_2 \cdots a_l)P$, or in short $(\nu\tilde{a})P$. We skip specifying the input name, if it is not free in the following process, i.e., $a.P$ refers to $a(x).P$ where $x \notin \mathit{fn}(P)$. $\bar{a}.P$ refers to outputting a value on link a that can be elided, e.g., due to lack of relevance in discussion.

Codebases Agent definitions are of the form $A(x_1, \cdots, x_n) \triangleq P$. Let's denote the set of agent and sub-agent definitions with $\mathcal{D}$. We assume the existence of a *universal codebase* $\mathcal{C}_{\mathcal{U}}$ consisting of agent definitions of such form. This codebase is used to define *top-level agents*. A top-level agent corresponds to a concurrent components of the system. Top-level agents are supposed to execute in parallel and occasionally communicate with each other to accomplish their own tasks, and in aggregate the concurrent system. Let $\mathcal{A}_{\mathcal{U}}$ be the set of top-level agent names such that $\mathcal{A}_{\mathcal{U}} \subset \mathcal{A}$. Throughout the paper we let m to be the size of $\mathcal{A}_{\mathcal{U}}$, comprising of $A_1, \cdots, A_m$. $\mathcal{C}_{\mathcal{U}}$ is defined as a function from top-level agent names to their definitions, i.e., $\mathcal{C}_{\mathcal{U}} : \mathcal{A}_{\mathcal{U}} \to \mathcal{D}$.

Moreover, we assume the existence of a *local codebase* for each top-level agent, denoted by $\mathcal{C}_{\mathcal{L}}(A)$ for top-level agent A. A local codebase consists of sub-agent (sub-process) definitions of the form $B^A(x_1, \cdots, x_n) \triangleq P$, where B is a sub-agent identifier, and A is a top-level agent identifier annotated in the definition of B. We treat sub-agents as internal modules or functions of a top-level process. Annotation of top-level agent identifier is used for this purpose, i.e., B^A specifies that B is a module of top-level agent A. The set of sub-agent names is denoted by $\mathcal{A}_{\mathcal{L}}$, defined as $\mathcal{A} - \mathcal{A}_{\mathcal{U}}$. $\mathcal{C}_{\mathcal{L}}$ is defined as the function with signature $\mathcal{C}_{\mathcal{L}} : \mathcal{A}_{\mathcal{U}} \to \mathcal{A}_{\mathcal{L}} \to \mathcal{D}$.

Note that any process and subprocess definition can be recursive, e.g., if $\mathcal{C}_{\mathcal{U}}(A) = [A(x_1, \cdots, x_n) \triangleq P]$ then $A(y_1, \cdots, y_n)$ may appear in P. In the following, we use A and B to range over top-level agent identifiers and sub-agent identifiers, resp. We use C to range over both top-level agents, A, and sub-agents, B^A. In the rest of the paper, we refer to top-level agents simply by "agents". We assume that in any (sub)process definition $C(x_1, \cdots, x_n) \triangleq P$, we have $\mathit{fn}(P) \subseteq \{x_1, \cdots, x_n\}$. This ensures that (sub)processes are closed.

Initial system Let $\mathcal{A}_{\mathcal{U}} = \{A_1, \cdots, A_m\}$. We posit a sequence of links $\tilde{c}$, that connect these agents in the system. Then the initial concurrent system $\mathfrak{s}$ is defined as

$$\mathfrak{s} ::= \langle P_{\mathfrak{s}}, \mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rangle, \tag{2.1}$$

where $P_{\mathfrak{s}} = (\nu\tilde{c})(A_1(\tilde{x}_1) \mid A_2(\tilde{x}_2) \mid \cdots \mid A_m(\tilde{x}_m))$, assuming $\mathit{fn}(P_{\mathfrak{s}}) = \emptyset$, i.e., $\bigcup_i \tilde{x}_i \subseteq \tilde{c}$.

Configurations We define system configurations as $\kappa ::= P$, where P is the process associated with the whole system. The initial configuration is then defined as $\kappa_0 = P_{\mathfrak{s}}$.

Substitutions A substitution is a function $\sigma : \mathcal{N} \to \mathcal{N}$. The notation $\{y/x\}$ is used to refer to a substitution that maps x to y, and acts as the identity function otherwise. $\{\tilde{y}/\tilde{x}\}$ is used to denote multiple explicit mappings in a substitution, where $\tilde{x}$ and $\tilde{y}$ are equal in length. $P\sigma$ refers to replacing free names in P according to σ. This is associated with renaming of bound names in P to avoid name clashes.

2.1.2 Semantics

In the following, we define evaluation contexts and the structural congruence between processes. These definitions facilitate the specification of unlabeled operational semantics in a concise manner.

Evaluation contexts A context is a process with a hole. An evaluation context $\mathcal{E}$ is a context whose hole is not under input/output prefix, i.e.,

$$\mathcal{E} ::= [\,] \mid (\mathcal{E}|P) \mid (P|\mathcal{E}) \mid (\nu a)\mathcal{E}.$$

Structural congruence Two processes P and Q are structurally congruent under the universal codebase $\mathcal{C}_\mathcal{U}$, denoted by $\mathcal{C}_\mathcal{U} \rhd P \equiv Q$ according to the following rules.

(i) Structural congruence is an equivalence relation.

(ii) Structural congruence is closed by the application of $\mathcal{E}$, i.e., $\mathcal{C}_\mathcal{U} \rhd P \equiv Q$ implies $\mathcal{C}_\mathcal{U} \rhd \mathcal{E}[P] \equiv \mathcal{E}[Q]$.

(iii) If P and Q are α-convertible, then $\mathcal{C}_\mathcal{U} \rhd P \equiv Q$.

(iv) The set of processes is an Abelian semigroup under | operator and unit element $\mathbf{0}$, i.e., for any $\mathcal{C}_\mathcal{U}$, P, Q, and R, we have $\mathcal{C}_\mathcal{U} \rhd P|\mathbf{0} \equiv P$, $\mathcal{C}_\mathcal{U} \rhd P|Q \equiv Q|P$, and $\mathcal{C}_\mathcal{U} \rhd P|(Q|R) \equiv (P|Q)|R$.

(v) For all $A \in \mathcal{A}_\mathcal{U}$, if $\mathcal{C}_\mathcal{U}(A) = [A(\tilde{x}) \triangleq P]$, then $\mathcal{C}_\mathcal{U} \rhd A(\tilde{y}) \equiv P\{\tilde{y}/\tilde{x}\}$.

(vi) $\mathcal{C}_\mathcal{U} \rhd (\nu a)\mathbf{0} \equiv \mathbf{0}$.

(vii) If $a \notin fn(P)$, then $\mathcal{C}_\mathcal{U} \rhd (\nu a)(P|Q) \equiv P|(\nu a)Q$.

(viii) $\mathcal{C}_\mathcal{U} \rhd (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$.

We may elide $\mathcal{C}_\mathcal{U}$ in the specification of the structural congruence, if it is clear from the context.

Operational semantics We define unlabeled reduction system in Figure 1, using judgment $\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \rhd \kappa \longrightarrow \kappa'$. We may elide $\mathcal{C}_\mathcal{U}$ and $\mathcal{C}_\mathcal{L}$ in the specification of reduction steps, since they are static and may be clear from the context, i.e., $\kappa \longrightarrow \kappa'$.

Note that according to structural congruence rules an agent invocation is structurally congruent to its definition (part v), and thus considered as an "implicit" step of execution according to rule STRUCT. Contrarily, rule CALL defines an "explicit" reduction step for sub-agent invocations. This is due to some technicality in our modeling: invocation of sub-agents could be logging preconditions and/or logging events (introduced in Section 2.2), and hence need special semantic treatment at the time of call (discussed later in Sections 2.3 and 2.4), e.g., deciding whether a record must be stored in the log.

For a (potentially infinite) system trace $\tau = \kappa_0\kappa_1\cdots$, we use notation $\mathcal{C}_\mathcal{U}, \mathcal{C}_\mathcal{L} \blacktriangleright \tau$ to specify the generation of trace τ under the universal and local codebases $\mathcal{C}_\mathcal{U}$ and

STRUCT
$$\frac{\mathcal{C}_{\mathcal{U}} \rhd P \equiv P' \qquad \mathcal{C}_{\mathcal{U}} \rhd Q \equiv Q' \qquad \mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd P \longrightarrow Q}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd P' \longrightarrow Q'}$$

CONTEXT
$$\frac{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd P \longrightarrow Q}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd \mathcal{E}[P] \longrightarrow \mathcal{E}[Q]}$$

CALL
$$\frac{\mathcal{C}_{\mathcal{L}}(A)(B) = [B^A(\tilde{x}) \triangleq P]}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd B^A(\tilde{y}) \longrightarrow P\{\tilde{y}/\tilde{x}\}}$$

COMM
$$\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd a(x).P \mid \bar{a}b.Q \longrightarrow P\{b/x\} \mid Q$$

Figure 1: Unlabeled reduction semantics of Π.

$\mathcal{C}_{\mathcal{L}}$, and according to the aforementioned unlabeled reduction system, i.e., $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd \kappa_i \longrightarrow \kappa_{i+1}$ for all $i \in \{0, 1, \cdots\}$.

For a system trace $\tau = \kappa_0\kappa_1\cdots$, system $\mathfrak{s}$ generates τ, denoted by $\mathfrak{s} \Downarrow \tau$ iff $\mathfrak{s}$ is defined as (2.1), κ_0 is defined as $P_{\mathfrak{s}}$, and $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd \kappa_i \longrightarrow \kappa_{i+1}$ for all $i \in \{0, 1, \cdots\}$.

toFOL$(\cdot)$ **instantiation for traces** In order to specify a trace logically, we need to instantiate *toFOL*$(\cdot)$ according to Definition 1.11. We consider the following predicates to logically specify a trace: Comm/3, Call/4, Context/2, UniversalCB/3, and LocalCB/4.[4]

Let $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \blacktriangleright \tau$, and $\tau = \kappa_0 \cdots \kappa\kappa' \cdots$. Moreover, let t denote a timing counter. We define a function that logically specifies a configuration within a trace. To this end, let the helper function *toFOL*(κ, t) return the logical specification of κ at time t. Essentially, *toFOL*(κ, t) specifies what the evaluation context and the redex are within κ at time t, defined as follows:

1. *toFOL*$(\kappa, t) = \{\mathrm{Comm}(t, a(x).P, \bar{a}b.Q), \mathrm{Context}(t, \mathcal{E})\}$,[5]if $\kappa \equiv \mathcal{E}[a(x).P \mid \bar{a}b.Q]$ and $\kappa' \equiv \mathcal{E}[P\{b/x\} \mid Q]$.

2. *toFOL*$(\kappa, t) = \{\mathrm{Call}(t, A, B, \tilde{y}), \mathrm{Context}(t, \mathcal{E})\}$, if $\kappa \equiv \mathcal{E}[B^A(\tilde{y}))]$ and $\kappa' \equiv \mathcal{E}[P\{\tilde{y}/\tilde{x}\}]$. Note that in Call$(t, A, B, \tilde{y})$, we treat $\tilde{y}$ as a single list of elements, rather than a sequence of elements passed as parameters to Call, i.e., Call is always a quaternary predicate.

As an example, consider α-converted structurally equivalent processes. Let $\kappa = a(x).(\nu b)\bar{x}b.\mathbf{0}|\bar{a}b.\mathbf{0}$. Since $\kappa \equiv \kappa' = a(x).(\nu d)\bar{x}d.\mathbf{0}|\bar{a}b.\mathbf{0}$, and $\kappa' \longrightarrow (\nu d)\bar{b}d.\mathbf{0}|\mathbf{0}$, we have $\kappa \longrightarrow (\nu d)\bar{b}d.\mathbf{0}|\mathbf{0}$ according to the rule STRUCT in Figure 1. Then, *toFOL*$(\kappa, t) =$ *toFOL*$(\kappa', t) = \{\mathrm{Comm}(t, a(x).(\nu d)\bar{x}d.\mathbf{0}, \bar{a}b.\mathbf{0}), \mathrm{Context}(t, [\,])\}$.

Logical specification of universal and local codebases, denoted by $\langle \mathcal{C}_{\mathcal{U}} \rangle$ and $\langle \mathcal{C}_{\mathcal{L}} \rangle$ resp., are defined as

1. $\langle \mathcal{C}_{\mathcal{U}} \rangle = \{\mathrm{UniversalCB}(A, \tilde{x}, P) \mid \mathcal{C}_{\mathcal{U}}(A) = [A(\tilde{x}) \triangleq P]\}$

2. $\langle \mathcal{C}_{\mathcal{L}} \rangle = \{\mathrm{LocalCB}(A, B, \tilde{x}, P) \mid \mathcal{C}_{\mathcal{L}}(A)(B) = [B^A(\tilde{x}) \triangleq P]\}$

[4] $/n$ refers to the arity of the predicate.

[5] Processes and evaluation contexts appear as predicate arguments in this presentation to boost readability. Note that their syntax can be written as string literals to comply with the syntax of predicate logic.

Note that in UniversalCB$(A, \tilde{x}, P)$ and LocalCB$(A, B, \tilde{x}, P)$, $\tilde{x}$ is a single list of elements, rather than a sequence of elements passed as parameters to the predicates, and thus these predicates have fixed arities.

We define logical specification of traces both for finite and infinite cases according to the logical specification of configurations, and universal and local codebases, i.e., using $\mathit{toFOL}(\kappa, t)$, $\langle \mathcal{C}_{\mathcal{U}} \rangle$, and $\langle \mathcal{C}_{\mathcal{L}} \rangle$. Let $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \blacktriangleright \tau$. If τ is finite, i.e., $\tau = \kappa_0 \kappa_1 \cdots \kappa_n$ for some n, then its logical specification is defined as $\mathit{toFOL}(\tau) = \bigcup_{i=0}^{n} \mathit{toFOL}(\kappa_i, i) \bigcup \langle \mathcal{C}_{\mathcal{U}} \rangle \bigcup \langle \mathcal{C}_{\mathcal{L}} \rangle$. Otherwise, for infinite trace $\tau = \kappa_0 \kappa_1 \cdots$, $\mathit{toFOL}(\tau) = \bigcup_{\tau' \in \mathit{prefix}(\tau)} \mathit{toFOL}(\tau') \bigcup \langle \mathcal{C}_{\mathcal{U}} \rangle \bigcup \langle \mathcal{C}_{\mathcal{L}} \rangle$, where $\mathit{toFOL}(\tau') = \bigcup_{i=0}^{n} \mathit{toFOL}(\kappa_i, i)$, for $\tau' = \kappa_0 \kappa_1 \cdots \kappa_n$. It is straightforward to show that $\mathit{toFOL}(\tau)$ is injective and monotonically increasing.

2.2 A Class of Logging Specifications

We define the class of logging specifications $\mathcal{LS}_{\mathit{call}}$ that specify temporal relations among module invocations in concurrent systems. $\mathcal{LS}_{\mathit{call}}$ is the set of all logging specifications LS defined as $\mathit{spec}(\Gamma_G, \{\text{LoggedCall}\})$, where Γ_G is a set of FOL formulas, called *guidelines*, including formulas of the form

$$\begin{aligned}
&\forall t_0, \cdots, t_n, \mathit{xs}_0, \cdots, \mathit{xs}_n \, . \\
&\quad \text{Call}(t_0, A_0, B_0, \mathit{xs}_0) \bigwedge_{i=1}^{n} \big(\text{Call}(t_i, A_i, B_i, \mathit{xs}_i) \wedge t_i < t_0\big) \wedge \varphi(t_0, \cdots, t_n) \wedge \varphi'(\mathit{xs}_0, \cdots, \mathit{xs}_n) \\
&\quad \bigwedge_{j=1}^{m} \big(\forall t'_j, \mathit{ys}_j \, . \, \psi_j(\mathit{xs}_0, \cdots, \mathit{xs}_n, \mathit{ys}_j) \wedge \psi'_j(t_0, \cdots, t_n, t'_j) \implies \neg\text{Call}(t'_j, A'_j, B'_j, \mathit{ys}_j)\big) \\
&\quad \implies \text{LoggedCall}(A_0, B_0, \mathit{xs}_0)
\end{aligned} \tag{2.2}$$

in which for all $l \in \{0, \cdots, n\}$, $A_l \in \mathcal{A}_{\mathcal{U}}$, $B_l \in \mathcal{A}_{\mathcal{L}}$, xs_l is a placeholder for a list of parameters passed to B_l, and Call$(t_l, A_l, B_l, \mathit{xs}_l)$ specifies the event of invoking module (subprocess) B_l by the top-level process A_l at time t_l with parameters xs_l. In addition, for all $k \in \{0, \cdots, m\}$, $A'_k \in \mathcal{A}_{\mathcal{U}}$, $B'_k \in \mathcal{A}_{\mathcal{L}}$, ys_k is a placeholder for a list of parameters passed to B'_k, and Call$(t_k, A'_k, B'_k, \mathit{ys}_k)$ specifies the event of invoking module (subprocess) B'_k by the top-level process A'_k at time t_k with parameters ys_k. In (2.2), $\varphi(t_0, \cdots, t_n)$ and $\psi'_j(t_0, \cdots, t_n, t'_j)$ are assumed to be a possibly empty conjunctive sequence of literals of the form $t_i < t_j$ for some i and j. Moreover, we define *(positive/negative) triggers* and *logging events* as follows: $\mathit{PostiveTriggers}(\mathit{LS}) = \{(A_1, B_1), \cdots, (A_n, B_n)\}$, $\mathit{NegativeTriggers} = \{(A'_1, B'_1), \cdots, (A'_m, B'_m)\}$, $\mathit{Triggers} = \mathit{PostiveTriggers} \cup \mathit{NegativeTriggers}$, and $\mathit{Logevent}(\mathit{LS}) = (A_0, B_0)$. *Logging preconditions* are predicates Call$(t_i, A_i, B_i, \tilde{x})$ for all $i \in \{1, \cdots, n\}$ and predicates Call$(t'_j, A'_j, B'_j, \tilde{y})$ for all $j \in \{1, \cdots, m\}$. As an additional condition, we assume that $\mathit{Logevent}(\mathit{LS}) \notin \mathit{Triggers}(\mathit{LS})$.

2.3 Target System Model

We define the target system model, denoted by Π_{log}, as an extension to Π with the following syntax and semantics. The instrumentation algorithm's job is to map a system specified in Π to a system in Π_{log}.

2.3.1 Syntax

Π_{log} extends prefixes with

$$\alpha ::= \cdots \mid \mathsf{callEvent}(A, B, \tilde{x}) \mid \mathsf{emit}(A, B, \tilde{x}) \mid \mathsf{addPrecond}(x, A) \mid \mathsf{sendPrecond}(x, A).$$

$\tilde{x}$ is considered as a single list of names in $\mathsf{callEvent}$ and emit, so that they have fixed arities.

A configuration κ, in Π_{log}, is defined as the quintuple $\kappa ::= (t, P, \Delta, \Sigma, \Lambda)$, with the following details. t is a timing counter. P is the process associated with the whole concurrent system. Processes in Π_{log} are defined similar to Π, without any extensions. $\Delta(\cdot)$ is a mapping that receives an agent identifier A and returns the set of logical preconditions (to log) that denote the events transpired locally in that agent. That is, $\Delta(A)$ is a set of predicates of the form $\text{Call}(t, A, B, \tilde{x})$. $\Sigma(\cdot)$ is a mapping that receives an agent identifier A and returns the set of all logical preconditions that have taken place in the triggers, i.e., in all agents $A' \in \mathcal{A}_{\mathcal{U}}$, where $(A', B) \in \mathit{Triggers}$ for some $B \in \mathcal{A}_{\mathcal{L}}$. That is, $\Sigma(A)$ is a set of predicates of the form $\text{Call}(t, A', B, \tilde{x})$, where $(A', B) \in \mathit{Triggers}$. These preconditions are supposed to be gathered by A from other agents A', in order to decide whether to log an event. $\Lambda(\cdot)$ is a mapping that receives an agent identifier A and returns the audit log recorded by that agent. $\Lambda(A)$ is a set of predicates of the form $\text{LoggedCall}(A, B, \tilde{x})$. The initial configuration is $\kappa_0 = (0, P_{\mathfrak{s}}, \Delta_0, \Sigma_0, \Lambda_0)$, where for any $A \in \mathcal{C}_{\mathcal{U}}$, $\Delta_0(A) = \Sigma_0(A) = \Lambda_0(A) = \emptyset$.

2.3.2 Semantics

We use judgment $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \kappa \longrightarrow \kappa'$ to specify a step of reduction in Π_{log}. Figure 2 depicts the unlabeled reduction semantics of Π_{log}. $\mathcal{C}_{\mathcal{U}}$, $\mathcal{C}_{\mathcal{L}}$, and Γ_G may be elided in the specification of reduction steps since they are static and may be clear from the context.

Π_{log} inherits the reduction semantics of Π, according to rule PI. Rule CALL_EV gives the reduction with prefix $\mathsf{callEvent}(A, B, \tilde{x})$. In this case, Δ gets updated for agent A with information about the invocation of subprocess B^A. In rule ADD_PRECOND, reduction with the prefix $\mathsf{addPrecond}(x, A)$ is specified. In this case, x is added to Σ. Rule SEND_PRECOND is about the reduction with prefix $\mathsf{sendPrecond}(x, A)$. In this case, the set of logging preconditions that are collected by A, i.e., $\Delta(A)$, is converted to a transferable object (aka object serialization), e.g., a string of characters describing the content of $\Delta(A)$, and sent though link x. Let `serialize()` be the semantic function that handles this conversion. With prefix $\mathsf{emit}(A, B, \tilde{x})$, agent A is supposed to study whether the predicate $\text{LoggedCall}(A, B, \tilde{x})$ is logically derivable from the

PI

$$\frac{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}} \rhd P \longrightarrow Q}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd (t, P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, Q, \Delta, \Sigma, \Lambda)}$$

CALL_EV

$$\frac{\Delta'(A) = \Delta(A) \cup \{\mathrm{Call}(t, A, B, \tilde{x})\} \qquad \forall A' \in \mathcal{A}_{\mathcal{U}} - \{A\}.\Delta'(A') = \Delta(A')}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \quad (t, \mathsf{callEvent}(A, B, \tilde{x}).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta', \Sigma, \Lambda)}$$

ADD_PRECOND

$$\frac{\Sigma'(A) = \Sigma(A) \cup \{x\} \qquad \forall A' \in \mathcal{A}_{\mathcal{U}} - \{A\}.\Sigma'(A') = \Sigma(A')}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \quad (t, \mathsf{addPrecond}(x, A).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta, \Sigma', \Lambda)}$$

SEND_PRECOND

$$\frac{y = \mathtt{serialize}(\Delta(A))}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \quad (t, \mathsf{sendPrecond}(x, A).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, \bar{x}y.P, \Delta, \Sigma, \Lambda)}$$

LOG

$$\frac{\begin{array}{c}\Sigma(A) \cup \Delta(A) \cup \Gamma_G \vdash \mathrm{LoggedCall}(A, B, \tilde{x}) \\ \Lambda'(A) = \Lambda(A) \cup \{\mathrm{LoggedCall}(A, B, \tilde{x})\} \qquad \forall A' \in \mathcal{A}_{\mathcal{U}} - \{A\}.\Lambda'(A') = \Lambda(A')\end{array}}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd (t, \mathsf{emit}(A, B, \tilde{x}).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta, \Sigma, \Lambda')}$$

NO_LOG

$$\frac{\Sigma(A) \cup \Delta(A) \cup \Gamma_G \nvdash \mathrm{LoggedCall}(A, B, \tilde{x})}{\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd (t, \mathsf{emit}(A, B, \tilde{x}).P, \Delta, \Sigma, \Lambda) \longrightarrow (t+1, P, \Delta, \Sigma, \Lambda)}$$

Figure 2: Unlabeled reduction semantics of $\Pi_{\log}$.

local set of preconditions, i.e., $\Delta(A)$, the set of preconditions that are collected by other agents involved in the enforcement of the logging specification, i.e., $\Sigma(A)$, and the set of guidelines Γ_G. If the predicate is derivable, then it is added to the audit log of A, i.e., $\Lambda(A)$. Otherwise, the log does not change. Rule LOG specifies the former case, whereas the rule NO_LOG specifies the latter.

For a (potentially infinite) system trace $\tau = \kappa_0\kappa_1 \cdots$, we use notation $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \blacktriangleright \tau$ to specify the generation of trace τ under the universal codebase $\mathcal{C}_{\mathcal{U}}$, local codebase $\mathcal{C}_{\mathcal{L}}$, and set of guidelines Γ_G, according to the reduction system, i.e., $\mathcal{C}_{\mathcal{U}}, \mathcal{C}_{\mathcal{L}}, \Gamma_G \rhd \kappa_i \longrightarrow \kappa_{i+1}$ for all $i \in \{0, 1, \cdots\}$.

The generated trace in $\Pi_{\log}$ out of a target system $\mathfrak{s}$, i.e., $\mathfrak{s} \Downarrow \tau$, can be defined in the same style as defined in Π, i.e., by some valid initial system in $\Pi_{\log}$[6], the initial configuration κ_0 in $\Pi_{\log}$, and the aforementioned reduction system for $\Pi_{\log}$.

The residual log of a configuration is defined as $\mathit{logof}(\kappa) = \mathbb{L} = \bigcup_{A \in \mathcal{A}_{\mathcal{U}}} \Lambda(A)$, where $\kappa = (\_, \_, \_, \_, \Lambda)$.[7] This instantiates $\tau \rightsquigarrow \mathbb{L}$ for $\Pi_{\log}$ (Definition 1.4). Since $\mathbb{L}$ is a set of logical literals, it suffices to define $\mathit{toFOL}(\cdot)$ for audit logs as $\mathit{toFOL}(\mathbb{L}) = \mathbb{L}$, which completes the instantiation of $\lfloor \mathbb{L} \rfloor$ (Definition 1.11).

Note that arbitrary systems in $\Pi_{\log}$ do not guarantee any correctness of audit logging. However, there is a subset of systems in $\Pi_{\log}$ that provably satisfy this property. These systems use the extended prefixes (introduced as part of $\Pi_{\log}$ syntax) in a particular way for this purpose. In the following section, we introduce an instrumen-

[6] In Section 2.4 one such initial system is given by the instrumentation algorithm.

[7] Underscore is used as wildcard.

tation algorithm to map any system in Π to a system in $\Pi_{\log}$, and later prove that any instrumented system satisfies correctness results for audit logging.

2.4 Instrumentation Algorithm

Instrumentation algorithm $\mathcal{I}$ takes a Π system, defined in (2.1), and a logging specification $LS \in \mathcal{LS}_{call}$, defined in Section 2.2, and produces a system $\mathfrak{s}'$ in $\Pi_{\log}$ defined as $\mathfrak{s}' = \left\langle P'_{\mathfrak{s}}, \mathcal{C}'_{\mathcal{U}}, \mathcal{C}'_{\mathcal{L}} \right\rangle$, where $P'_{\mathfrak{s}} = (\nu\tilde{c})(\nu\tilde{c}')(A_1(\tilde{x}'_1) \mid A_2(\tilde{x}'_2) \mid \cdots \mid A_m(\tilde{x}'_m))$. $\tilde{c}'$ is the sequence of names of the form c_{ij} which are all fresh, i.e., they are not used already in (2.1). Moreover, it is assumed that sub-agent identifiers D_{ij} are also fresh, i.e., they are undefined in $\mathcal{C}_{\mathcal{L}}$ component of (2.1).

Intuitively, $\mathcal{I}$ works as follows.

(i) $\mathcal{I}$ adds new links c_{ij} between agents A_i and A_j, where A_i is the agent that includes a sub-agent whose invocation is considered a logging event, and A_j is some agent that includes a sub-agent whose invocation is a trigger for that logging event. c_{ij} is used as a link between A_i and A_j to communicate logging preconditions (by sendPrecond and addPrecond prefixes).

(ii) Regarding the invocation of a sub-agent B^A,

  (a) if the invocation of B^A is a trigger, then the execution of B^A must be preceded by callEvent prefix. This way, the invocation of B^A is stored in A's local set of logging precondition ($\Delta(A)$), according to the rule CALL_EV.

  (b) if the invocation of B^A is a logging event, then execution of B^A must be preceded by callEvent, similar to the case above. Next, it must communicate on appropriate links (c_{ij}s) with all other agents that are involved as triggers according to the logging specification. To this end, B^A is supposed to notify each of those agents to send their collected preconditions. After receiving all those preconditions from involved agents on the dedicated links, it adds them to $\Sigma(A)$. This is done using addPrecond prefixes, according to the rule ADD_PRECOND. Then, it studies whether the invocation must be logged, before following normal execution. This is facilitated by emit prefix (rules LOG and NO_LOG).

  (c) if the invocation of B^A is neither a trigger nor a logging event, then that sub-agent executes without any change in behavior.

(iii) Regarding the invocation of an agent A

  (a) if A includes a sub-agent B^A whose invocation is considered a trigger, then A must be able to receive and handle incoming requests for collected preconditions. This is done by adding a subprocess to A that always listens for requests on the dedicated link (c_{ij}) between itself and the agent that may send such requests. Upon receiving such a request, it sends back the preconditions, handled by prefix $\mathsf{sendPrecond}$ according to the rule SEND_PRECOND, and then continues to listen on the link.

(b) if A does not include any trigger invocation of a sub-agent, then A executes without any changes.

Formally, the details of the returned system $\mathfrak{s}'$ are as follows:

(i) $\tilde{c}'$: is the sequence of all names c_{ij} where $(A_i, B) = \mathit{Logevent}(LS)$ for some $B \in \mathcal{A}_{\mathcal{L}}$, and $(A_j, B') \in \mathit{Triggers}(LS)$ for some $B' \in \mathcal{A}_{\mathcal{L}}$.

(ii) $\mathcal{C}'_{\mathcal{L}}$:

(a) $\mathcal{C}'_{\mathcal{L}}(A)(B) = [B^A(\tilde{x}) \triangleq \mathsf{callEvent}(A, B, \tilde{x}).P]$, if $(A, B) \in \mathit{Triggers}(LS)$ and $\mathcal{C}_{\mathcal{L}}(A)(B) = [B^A(\tilde{x}) \triangleq P]$.

(b) $\mathcal{C}'_{\mathcal{L}}(A_0)(B_0) = [B_0^{A_0}(\tilde{x}) \triangleq \mathsf{callEvent}(A_0, B_0, \tilde{x}) \,.\, \bar{c_{01}} \,.\, \cdots \,.\, \bar{c_{0n}} \,.\, c_{01}(p_1) \,.\, \cdots \,.\, c_{0n}(p_n) \,.$ $\mathsf{addPrecond}(p_1, A_0). \cdots \,.\, \mathsf{addPrecond}(p_n, A_0) \,.\, \mathsf{emit}(A_0, B_0, \tilde{x}).P]$,
if $(A_0, B_0) = \mathit{Logevent}(LS), \mathcal{C}_{\mathcal{L}}(A_0)(B_0) = [B_0^{A_0}(\tilde{x}) \triangleq P]$, and $\mathit{Triggers}(LS) = \{(A_1, B_1), \cdots, (A_n, B_n)\}$.

(c) $\mathcal{C}'_{\mathcal{L}}(A)(B) = \mathcal{C}_{\mathcal{L}}(A)(B)$, otherwise.

(iii) $\mathcal{C}'_{\mathcal{U}}$:

(a) If $(A_j, B) \in \mathit{Triggers}(LS)$ for some $B \in \mathcal{A}_{\mathcal{L}}$, and A_i be the agent that $(A_i, B') = \mathit{Logevent}(LS)$ for some $B' \in \mathcal{A}_{\mathcal{L}}$, then $\mathcal{C}'_{\mathcal{U}}(A_j) = [A_j(\tilde{x}, c_{ij}) \triangleq P|D_{ij}^{A_j}(c_{ij})]$, where $\mathcal{C}_{\mathcal{U}}(A_j) = [A_j(\tilde{x}) \triangleq P]$, and $\mathcal{C}'_{\mathcal{L}}(A_j)(D_{ij}) = [D_{ij}^{A_j}(c_{ij}) \triangleq c_{ij}.\mathsf{sendPrecond}(c_{ij}, A_j).D_{ij}^{A_j}(c_{ij})]$.

(b) If $(A_j, B) \notin \mathit{Triggers}(LS)$ for any $B \in \mathcal{A}_{\mathcal{L}}$, then $\mathcal{C}'_{\mathcal{U}}(A_j) = \mathcal{C}_{\mathcal{U}}(A_j)$.

Note that D_{ij} is defined recursively to facilitate listening on c_{ij} indefinitely for incoming requests about logging preconditions. In addition, since c_{ij} is fresh, $c_{ij} \notin \mathit{fn}(P)$. Therefore, P cannot communicate on this link, e.g., to compromise logging attempts.

2.4.1 Instantiation of :≈

According to Definition 1.5, semantics preservation relies on an abstraction of correspondence relation :≈ between source and target traces. In this section, we instantiate this relation for $\mathcal{I}$. We define the source and target trace correspondence relation as follows: $\tau_1\kappa_1 :\approx \tau_2\kappa_2$ iff $\kappa_1 = P_1$, $\kappa_2 = (t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2)$, and $\mathit{trim}(P_2) = P_1$. Function *trim* is formally defined in Figure 3. Intuitively, it removes all prefixes, sub-agents, and link names that $\mathcal{I}$ may add to a process.

$$\mathit{trim}(\mathbf{0}) = \mathbf{0} \qquad \mathit{trim}(\alpha.P) = \begin{cases} \mathit{trim}(P) & \text{if } \alpha = c_{ij}, \bar{c_{ij}}, c_{ij}(x), \bar{c_{ij}}x, \\ & \mathsf{callEvent}(A, B, \tilde{x}), \mathsf{addPrecond}(x, A), \\ & \mathsf{sendPrecond}(x, A), \mathsf{emit}(A, B, \tilde{x}) \\ \alpha.\mathit{trim}(P) & \text{otherwise} \end{cases}$$

$$\mathit{trim}(P|Q) = \begin{cases} \mathit{trim}(P) & \text{if } Q = D^A_{ij} \text{ for some } i, j, A \\ \mathit{trim}(Q) & \text{if } P = D^A_{ij} \text{ for some } i, j, A \\ \mathit{trim}(P)|\mathit{trim}(Q) & \text{otherwise} \end{cases}$$

$$\mathit{trim}((\nu x)P) = \begin{cases} \mathit{trim}(P) & \text{if } x = c_{ij} \text{ for some } i, j \\ (\nu x)\mathit{trim}(P) & \text{otherwise} \end{cases} \qquad \mathit{trim}(C(\tilde{x})) = \begin{cases} \mathit{trim}(C(\tilde{y})) & \text{if } \tilde{x} = \tilde{y}, c_{ij} \text{ for some } i, j \\ C(\tilde{x}) & \text{otherwise} \end{cases}$$

Figure 3: Function *trim*.

2.4.2 Main Results

Main properties include three results. The instrumentation algorithm $\mathcal{I}$ is semantics preserving, deadlock-free, correct. These are specified in Theorems 2.7, 2.8, and 2.13, resp.

Lemma 2.1. $P_{\mathfrak{s}} :\approx (0, P'_{\mathfrak{s}}, \Delta_0, \Sigma_0, \Lambda_0)$.

Proof. It is straightforward to show that $\mathit{trim}(P'_{\mathfrak{s}}) = P_{\mathfrak{s}}$, according to the definition of *trim* in Figure 3. □

Let $\longrightarrow^*$ be the reflexive and transitive closure of reduction relation $\longrightarrow$.

Lemma 2.2. *Let* $(t, \mathcal{E}[\mathit{trim}(P)], \Delta, \Sigma, \Lambda) \longrightarrow (t', \mathcal{E}[P'], \Delta', \Sigma', \Lambda')$*. Then, there exists some* t''*,* P''*,* Δ''*,* Σ''*, and* Λ''*, where*

- $(t, \mathcal{E}[P], \Delta, \Sigma, \Lambda) \longrightarrow^* (t'', \mathcal{E}[P''], \Delta'', \Sigma'', \Lambda'')$*, and*
- $\mathit{trim}(P') = \mathit{trim}(P'')$*.*

Proof. By induction on the structure of P. □

Lemma 2.3. *Let* $\tau_1 P_1 :\approx \tau_2(t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2)$ *and* $P_1 \longrightarrow Q_1$*. Then, there exists some non-trivial trace* τ'_2 *such that* $(t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2) \Downarrow \tau'_2$ *and* $\tau_1 P_1 Q_1 :\approx \tau_2 \tau'_2$*.*

Proof. By induction on the derivation of $P_1 \longrightarrow Q_1$, and application of Lemma 2.2. □

Lemma 2.4. *Let* $\tau_1 P_1 :\approx \tau_2(t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2)$ *and* $P_1 \Downarrow \tau'_1$*. Then, there exists some trace* τ'_2 *such that* $(t_2, P_2, \Delta_2, \Sigma_2, \Lambda_2) \Downarrow \tau'_2$ *and* $\tau_1 \tau'_1 :\approx \tau_2 \tau'_2$*.*

Proof. By induction on the derivation of $P_1 \Downarrow \tau'_1$, and application of Lemma 2.3. □

Lemma 2.5. *Let* $\tau_1 P_1 :\approx \tau_2 \kappa_2$ *and* $\kappa_2 \longrightarrow \kappa'_2$*. Then, there exists some trace* τ'_1 *such that* $P_1 \Downarrow \tau'_1$ *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 2.6. *Let $\tau_1 P_1 :\approx \tau_2 \kappa_2$ and $\kappa_2 \Downarrow \tau_2'$. Then, there exists some trace τ_1' such that $P_1 \Downarrow \tau_1'$ and $\tau_1 \tau_1' :\approx \tau_2 \tau_2'$.*

Proof. By induction on the derivation of $\kappa_2 \Downarrow \tau_2'$, and application of Lemma 2.5. □

Theorem 2.7 (Semantics preservation)**.** *$\mathcal{I}$ is semantics preserving (Definition 1.5).*

Proof. Lemmas 2.1, 2.4, and 2.6 entail the result. In essence, Lemmas 2.1 and 2.4 satisfy the first condition in Definition 1.5, whereas Lemmas 2.1 and 2.6 satisfy the second condition in that definition. □

Theorem 2.8 (Deadlock-freeness)**.** *$\mathcal{I}$ is deadlock-free (Definition 1.6).*

Proof. Let $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, and $\mathcal{I}(\mathfrak{s}, LS)$ be stuck following τ', i.e., for all configurations κ', we have $\mathcal{I}(\mathfrak{s}, LS) \not\Downarrow \tau'\kappa'$. According to Theorem 2.7, there exists some trace τ, such that $\mathfrak{s} \Downarrow \tau$ and $\tau :\approx \tau'$. Then, there are two possible cases:

1. For any configuration κ, $\mathfrak{s} \not\Downarrow \tau\kappa$. Then, according to Definition 1.6, $\mathcal{I}(\mathfrak{s}, LS)$ is deadlock-free, vacuously.

2. There exists some configuration κ such that $\mathfrak{s} \Downarrow \tau\kappa$. Then, according to Lemma 2.3, there exists some non-trivial trace τ'' such that $\mathit{tail}(\tau') \Downarrow \tau''$ and $\tau\kappa :\approx \tau'\tau''$. The former entails that $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'\tau''$, which contradicts with the assumption about $\mathcal{I}(\mathfrak{s}, LS)$ being stuck, and completes the proof.

□

Lemma 2.9. *Let $(t, P, \Delta, \Sigma, \Lambda) \longrightarrow (t', Q, \Delta', \Sigma', \Lambda')$. Moreover, assume that $\mathrm{LoggedCall}(A, B, \tilde{x}) \in \Lambda'(A) - \Lambda(A)$. Then, there exist some evaluation context $\mathcal{E}$ and some process R such that $P = \mathcal{E}[\mathsf{emit}(A, B, \tilde{x}).R]$.*

Proof. By induction on the derivation of unlabeled reduction relation, i.e., $(t, P, \Delta, \Sigma, \Lambda) \longrightarrow (t', Q, \Delta', \Sigma', \Lambda')$. □

Lemma 2.10. *Let $(t, P, \Delta, \Sigma, \Lambda) \Downarrow \tau(t', P', \Delta', \Sigma', \Lambda')$. Moreover, assume that $\mathrm{LoggedCall}(A, B, \tilde{x}) \in \Lambda'(A) - \Lambda(A)$. Then, there exist some evaluation context $\mathcal{E}$ and some process R along with trace τ' and configurations κ_1 and κ_2 such that $\tau'\kappa_1\kappa_2 \in \mathit{prefix}(\tau(t', P', \Delta', \Sigma', \Lambda'))$, and $P_1 = \mathcal{E}[\mathsf{emit}(A, B, \tilde{x}).R]$, where $\kappa_1 = (\_, P_1, \_, \_, \_)$.*

Proof. By induction on $(t, P, \Delta, \Sigma, \Lambda) \Downarrow \tau(t', P', \Delta', \Sigma', \Lambda')$ and application of Lemma 2.9. □

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

Lemma 2.11. *If $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, $\tau :\approx \tau'$, and $\tau' \rightsquigarrow \mathbb{L}$, then $\mathit{toFOL}(\mathbb{L}) \subseteq \mathit{Closure}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\mathrm{LoggedCall}\})$.*

Proof. Let $\mathit{tail}(\tau') = (\_, \_, \_, \_, \Lambda)$. Assume that $\text{LoggedCall}(A, B, \tilde{x}) \in \mathit{toFOL}(\mathbb{L})$. Then, according to Lemma 2.10, there exist some trace $\hat{\tau}$ and configurations κ_1 and κ_2 such that $\hat{\tau}\kappa_1\kappa_2 \in \mathit{prefix}(\tau')$, $\kappa_1 = (\_, P_1, \_, \_, \_)$, and $P_1 = \mathcal{E}[\mathsf{emit}(A, B, \tilde{x}).R]$. By Theorem 2.7, we know that there exist some source trace τ_0 such that $\mathfrak{s} \Downarrow \tau_0$, 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 $\mathcal{I}$ definition, prefix emit can only appear in the log event of a logging specification. Therefore, preconditions of rule (2.2) are satisfied by $\Gamma_G \cup \mathit{toFOL}(\tau_0)$. Due to $\tau_0 \in \mathit{prefix}(\tau)$ and monotonicity of $\mathit{toFOL}()$, the precondition of rule (2.2) are satisfied by $\Gamma_G \cup \mathit{toFOL}(\tau)$. Therefore, $\text{LoggedCall}(A, B, \tilde{x}) \in \mathit{Closure}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$. □

Lemma 2.12. *If $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, $\tau :\approx \tau'$, and $\tau' \rightsquigarrow \mathbb{L}$, then $\mathit{Closure}(\mathit{toFOL}(\mathbb{L})) \supseteq \mathit{Closure}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$.*

Proof. Note that $\mathit{Closure}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$ is a set of formulas with literals that have predicate symbol LoggedCall. Thus it suffices to show that if such literals are in $\mathit{Closure}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$, then they are in $\mathit{toFOL}(\mathbb{L})$.

Assume that $\text{LoggedCall}(A_0, B_0, \tilde{x}) \in \mathit{Closure}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$. This entails $\text{Call}(t, A_0, B_0, \tilde{xs}) \in \mathit{toFOL}(\tau)$ for some t in which preconditions of rule (2.2) are satisfied. Then, $\mathit{tail}([\cdots t]\tau) = \mathcal{E}[B_0^{A_0}(\tilde{xs})]$ for some $\mathcal{E}$. According to Theorem 2.7, there exists some target trace $\hat{\tau}$ such that $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \hat{\tau}$, and $[\cdots t]\tau :\approx \hat{\tau}$. Since $\mathcal{I}$ is deadlock-free (Theorem 2.8), we know that $\mathcal{I}(\mathfrak{s}, LS)$ is not stuck after following execution trace $\hat{\tau}$. Let $\mathit{tail}(\hat{\tau}) = (\_, \hat{P}, \_, \_, \_)$. Then, $\mathit{trim}(\hat{P}) = \mathcal{E}[B_0^{A_0}(\tilde{xs})]$. We have

$$(\hat{t}, \mathcal{E}[\mathit{trim}(\hat{P})], \hat{\Delta}, \hat{\Sigma}, \hat{\Lambda}) \longrightarrow (\hat{t} + 1, \mathcal{E}[\mathsf{callEvent}(A_0, B_0, \tilde{x}).\bar{c_{01}}. \cdots .\bar{c_{0n}}.c_{01}(p_1). \cdots .c_{0n}(p_n).$$
$$\mathsf{addPrecond}(p_1, A_0). \cdots .\mathsf{addPrecond}(p_n, A_0).\mathsf{emit}(A_0, B_0, \tilde{x}).R], \hat{\Delta}, \hat{\Sigma}, \hat{\Lambda})$$

By Lemma 2.2, we then have some trace τ_0, and configurations $\kappa_1, \cdots, \kappa_{3n+4}$ such that $(\hat{t}, \mathcal{E}[\hat{P}], \hat{\Delta}, \hat{\Sigma}, \hat{\Lambda}) \Downarrow \tau_0\kappa_0\kappa_1 \cdots \kappa_{3n+4}$, where

- $\kappa_1 = (\_, \mathcal{E}'[B_0^{A_0}(\tilde{x})], \_, \_, \hat{\Lambda})$,
- $\kappa_2 = (\_, \mathcal{E}'[\mathsf{callEvent}(A_0, B_0, \tilde{x}).\bar{c_{01}}. \cdots .\bar{c_{0n}}.c_{01}(p_1). \cdots .c_{0n}(p_n).$
  $\mathsf{addPrecond}(p_1, A_0). \cdots .\mathsf{addPrecond}(p_n, A_0).$
  $\mathsf{emit}(A_0, B_0, \tilde{x}).R], \_, \_, \hat{\Lambda})$,
- ...
- $\kappa_{3n+3} = (\_, \mathcal{E}'[\mathsf{emit}(A_0, B_0, \tilde{x}).R], \_, \_, \hat{\Lambda})$, and
- $\kappa_{3n+4} = (\_, \mathcal{E}'[R], \_, \_, \hat{\Lambda}')$.

Note that $\text{LoggedCall}(A_0, B_0, \tilde{x}) \in \hat{\Lambda}'(A_0)$, which entails that if $\hat{\tau}\tau_0\kappa_1 \cdots \kappa_{3n+4} \rightsquigarrow \hat{\mathbb{L}}$ then $\text{LoggedCall}(A_0, B_0, \tilde{x}) \in \hat{\mathbb{L}}$. Since $[\cdots (t + 1)]\tau :\approx \hat{\tau}\tau_0\kappa_1 \cdots \kappa_{3n+4}$ and $[\cdots (t + 1)]\tau \in \mathit{prefix}(\tau)$, 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 $\text{LoggedCall}(A_0, B_0, \tilde{x}) \in \mathbb{L}$, and thus $\text{LoggedCall}(A_0, B_0, \tilde{x}) \in \mathit{toFOL}(\mathbb{L})$. □

Theorem 2.13 (Instrumentation correctness). *$\mathcal{I}$ is correct (Definition 1.7).*

Proof. Let $\mathfrak{s}$ be a source system and LS be a logging specification defined as Section 2.2. If $\mathcal{I}(\mathfrak{s}, LS) \Downarrow \tau'$, $\tau :\approx \tau'$, and $\tau' \rightsquigarrow \mathbb{L}$, then we need to show that $\mathit{Closure}(\mathit{toFOL}(\mathbb{L})) = LS(\tau)$. By Lemma 2.11, we have $\mathit{toFOL}(\mathbb{L}) \subseteq \mathit{Closure}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\})$. This entails that

$$\begin{aligned}\mathit{Closure}(\mathit{toFOL}(\mathbb{L})) &\subseteq \mathit{Closure}(\mathit{Closure}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\}))\\ &= \mathit{Closure}(\mathit{Closure}(\mathit{Closure}(\Gamma_G) \cup \mathit{Closure}(\mathit{toFOL}(\tau))) \cap \mathit{FOL}(\{\text{LoggedCall}\}))\\ &= LS(\tau).\end{aligned}$$

In addition, by Lemma 2.12, we have

$$\begin{aligned}\mathit{Closure}(\mathit{toFOL}(\mathbb{L})) &\supseteq \mathit{Closure}(\mathit{Closure}(\Gamma_G \cup \mathit{toFOL}(\tau)) \cap \mathit{FOL}(\{\text{LoggedCall}\}))\\ &= \mathit{Closure}(\mathit{Closure}(\mathit{Closure}(\Gamma_G) \cup \mathit{Closure}(\mathit{toFOL}(\tau))) \cap \mathit{FOL}(\{\text{LoggedCall}\}))\\ &= LS(\tau).\end{aligned}$$

These two results imply that $\lfloor \mathbb{L} \rfloor = \mathit{Closure}(\mathit{toFOL}(\mathbb{L})) = LS(\tau)$. □

3 Conclusion

In this paper, we have proposed an implementation model to enforce correct audit logging in concurrent environments that support negative triggers. In essence, we have shown that the already specified algorithm for logging specifications that are restricted to positive triggers is able to instrument concurrent systems according to a formal specification of audit logging requirements that include negative triggers as well. We go beyond Horn clause logic to specify these logging requirements, which assert temporal relations among the events that transpire in different concurrent components of the system as wells events that should not take place. We have proven that our algorithm is semantics preserving, i.e., the instrumented system behaves similar to the original system, modulo operations that correspond to audit logging. Moreover, we have proven that our algorithm guarantees correct audit logs. This ensures that the instrumented system avoids missing any logging event, as well as logging unnecessary events. Correctness of audit logs are defined according to an information-algebraic semantic framework. In this semantic framework, information containment is used to compare the runtime behavior vs. the generated audit log.

References

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

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

[3] Amir-Mohammadian, S., Kari, C.: Correct audit logging in concurrent systems. Electronic Notes in Theoretical Computer Science 351, 115–141 (September 2020), part of Special Issue: Proceedings of LSFA 2020, the 15th International Workshop on Logical and Semantic Frameworks, with Applications

[4] Erlingsson, Ú.: The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University (2003)

[5] Hoare, C.A.R.: Communicating sequential processes. In: The origin of concurrent programming, pp. 413–443. Springer (1978)

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

[7] Milner, R.: Communication and concurrency, vol. 84. Prentice hall New York etc. (1989)

[8] Parrow, J.: An introduction to the π-calculus. In: Handbook of Process Algebra, pp. 479–543. Elsevier (2001)

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