

Semi-Automatic Synthesis of Security Policies by Invariant-Guided Abduction - Full version -

Clément Hurlin^{1,2} and Hélène Kirchner¹

¹ INRIA Bordeaux – Sud-Ouest Microsoft R&D France
351 avenue de la libération ² 39 quai du Président Roosevelt
33405 Talence, France 92130 Issy-les-Moulineaux, France

firstname.lastname@inria.fr

Abstract. We present a specification approach of secured systems as transition systems and security policies as constraints that guard the transitions. In this context, security properties are expressed as invariants. Then we propose an abduction algorithm to generate possible security policies for a given transition-based system. Because abduction is guided by invariants, the generated security policies enforce security properties specified by these invariants. In this framework we are able to tune abduction in two ways in order to: *(i)* filter out bad security policies and *(ii)* generate additional possible security policies. Invariant-guided abduction helps *designing* policies and thus allows using formal methods much earlier in the process of building secured systems. This approach is illustrated on role-based access control systems.

1 Introduction

Security administration in large, open, distributed and heterogeneous environments has strongly motivated research on security policy specification and analysis, but is yet nowadays a challenge. There is a great amount of proposals for security languages and frameworks for access control and more generally security policies, but even with an expressive policy language, a user may have difficulty understanding the overall effects of a policy and may not foresee the consequences of composing several policies. To improve confidence, both logic-based languages and verification techniques have been set up, as well as security analysis concepts. Our approach is in the line of logic-based languages providing a well-understood formalism, amenable to verification, proof and analysis.

We specify secured systems as transition systems whose states record the relevant security information in the current environment, and whose transitions model their changes. Security policy authorisations guard the transitions. Guards are constraints that are solved in the current state. The transition occur only when the constraint is satisfied and the changes may take into account the solutions of the constraint. In this context, the system and the policy that restrict it are given separately, but share a common language. Contrary to close related

work (such as TLA [16]), this modular design allows to plug-in different security policies and to compare them. Transition rules and constraints are expressed in first-order logic. Many security properties can then be expressed as invariants of the transition system and verified: invariants are first-order formulae that should hold in any state of the system restricted by the policy.

Our main contribution is to show how to synthesise security policies that enforce given invariants. Using *abduction*, the method consists in trying to prove the desired properties - invariants - and inspecting failed proof attempts to guess security policies. While current approaches on security analysis use formal methods to validate already existing security policies, invariant-guided abduction helps *designing* policies and thus allows using formal methods much earlier in the process of building secured systems. The modular design of system and policy described above is essential for the abduction algorithm, that requires that the system's states and transitions are defined *before* generating policies. Our approach uses first-order deduction: it can be implemented in frameworks using transition systems and whose proof obligations can be discharged with sequent calculus (such as TLA [16] or B [3]).

In Sec. 2, we define secured systems and security policies using standard first-order logic. In Sec. 3, we show how to prove invariants of secured systems inductively. In Sec. 4, we present our abduction algorithm that proposes possible security policies satisfying by construction such invariants. We also provide some heuristics for choosing between different possibilities, when any. In Sec. 5, the synthesis method is illustrated on a role-based access control (RBAC) system. Finally, we discuss related work and give further perspectives in Sec. 6.

2 Secured systems and security policies

In order to model security policies applied to a system, we choose to model the part of information in the system relevant to security as a state, the evolution of this information as a transition relation, and the policy as a constraint that restricts the transition relation.

For instance in a role-based access control (RBAC) [24, 12, 18] policy, the state is built from current information on users, sessions, roles, inheritance relation; the transition relation is defined from events like *create a session* or *assign a role* whose parameters are abstractly defined but should instantiate on actual users, sessions, roles... in the current state where the event occurs; the effect of a transition triggered by an event *create a session* is to modify the current state of information for instance by opening a session for a current user and giving him a list of actual roles. The security policy should constrain this transition to cases where the session is not yet opened and the actual roles are allowed for the current user in the current state where the event occurs.

In this approach the transition rules and the policy constraints are specified abstractly with first-order formulae but their evaluation is performed in a changing environment which is the current state of information in the system.

A specification \mathbb{SP} consists of a signature $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{P})$ with a set of *sorts* \mathcal{S} , a set of functions \mathcal{F} , a set of *predicates* \mathcal{P} , together with a set of Σ -formulae \mathbb{T} . Because in this paper, the functions purpose is only to build events, we require: (1) \mathcal{S} contains a sort *Event*, (2) functions and predicates do not have *Event* in their domains, and (3) functions all have *Event* as co-domain.

Lists of variables (x_1, \dots, x_n) are denoted simply \bar{x} , $f(\bar{x})$ (resp. $p(\bar{y})$) denotes $f(x_1, \dots, x_n)$ (resp. $p(y_1, \dots, y_m)$) when f is a function of Σ of arity n , (resp. p is a predicate of arity m), and $\bar{x} = \bar{x}'$ denotes component-wise equality of lists of variables.

For example, the specification of a RBAC system [24, 12, 18] can be given as follows:

$$\begin{aligned}
 \text{sorts } \mathcal{S} &\triangleq \left\{ \begin{array}{l} \textit{User}, \textit{Role}, \textit{Roles} \\ \textit{Session}, \textit{Event} \end{array} \right\} & \text{predicates } \mathcal{P} &\triangleq \left\{ \begin{array}{l} \textit{UR} : \textit{User} \times \textit{Role} \\ \textit{user} : \textit{Session} \times \textit{User} \\ \textit{role} : \textit{Session} \times \textit{Role} \\ \leq : \textit{Role} \times \textit{Role} \end{array} \right\} \\
 \text{functions } \mathcal{F} &\triangleq \left\{ \begin{array}{l} \textit{createSession} : \textit{User} \times \textit{Session} \times \textit{Roles} \rightarrow \textit{Event} \\ \textit{addActiveRole} : \textit{User} \times \textit{Session} \times \textit{Role} \rightarrow \textit{Event} \\ \textit{dropActiveRole} : \textit{User} \times \textit{Session} \times \textit{Role} \rightarrow \textit{Event} \\ \textit{deleteSession} : \textit{User} \times \textit{Session} \rightarrow \textit{Event} \\ \textit{assignRole} : \textit{User} \times \textit{Role} \rightarrow \textit{Event} \\ \textit{addInheritance} : \textit{Role} \times \textit{Role} \rightarrow \textit{Event} \end{array} \right\} \\
 \text{theory } \mathbb{T} &\triangleq \left\{ \begin{array}{l} \forall x. x \leq x \\ \forall x y z. x \leq y \wedge y \leq z \Rightarrow x \leq z \\ \forall x y. x \leq y \wedge y \leq x \Rightarrow x = y \\ \forall u r r'. \textit{UR}(u, r) \wedge r' \leq r \Rightarrow \textit{UR}(u, r') \end{array} \right\} \\
 \mathbb{SP}_{\text{RBAC}} &\triangleq ((\mathcal{S}, \mathcal{F}, \mathcal{P}), \mathbb{T})
 \end{aligned}$$

To keep the specification small, we omit objects and operations, as well as administrative commands and review functions [18]. Predicate $\textit{UR}(u, r)$ indicates that role r is assigned to user u ; $\textit{user}(s, u)$ (\textit{SU} in [18]) indicates that user u opened session s ; $\textit{role}(s, r)$ (\textit{SR} in [18]) indicates the set of roles activated by session s user; and \leq (\textit{INH} in [18]) specifies the hierarchy between roles. Functions specify events. The theory states that \leq is an order and axiomatizes the hierarchy of roles.

The states of the system are first-order \mathbb{SP} -interpretations \mathcal{I} that map sorts and predicates into sets. We refer to [8] for the precise definitions. Because in this paper, functions only represent events, they do not appear in formulae and we can leave them uninterpreted.

Fig. 1 shows an $\mathbb{SP}_{\text{RBAC}}$ -interpretation: $\mathcal{I}_{\text{RBAC}}$. It represents the state of a system where the set of users includes *Bob*, role *Secretary* is assigned to *Bob*, *Bob* opened session 0 and *Bob* associated roles *Secretary* and *Worker* with session 0.

$$\mathcal{I}_{RBAC} \triangleq \left\{ \begin{array}{l} User \mapsto \{Alice, Bob\} \\ Role \mapsto \{Secretary, Worker\} \\ Session \mapsto \mathbb{N} \\ UR \mapsto \{(Alice, Worker), (Bob, Secretary)\} \\ \leq \mapsto \{(Worker, Secretary)\} \\ user \mapsto \{(0, Bob)\} \\ role \mapsto \{(0, Secretary), (0, Worker)\} \end{array} \right\}$$

Fig. 1. Definition of a $\mathbb{S}\mathbb{P}_{RBAC}$ -interpretation: \mathcal{I}_{RBAC}

2.1 Transitions

The transition rules of the system describe the evolution of the states when specific events occur.

Definition 1 (Transition rule). A transition rule (or simply rule) over a specification $\mathbb{S}\mathbb{P}$ is a pair “ $f(\bar{x}) : action$ ” where:

1. $f(\bar{x})$ is called the pattern and variables in \bar{x} are called parameters;
2. action is a \bullet -separated list of atomic actions of the form $p(\bar{y}) \mid \phi$ or $\neg p(\bar{y}) \mid \phi$.
In an action $p(\bar{y}) \mid \phi$, some variables in \bar{y} may be parameters, and ϕ 's free variables must occur in $\bar{x} \cup \bar{y}$ (similarly for negated atomic actions). Given a transition rule r , we denote by pattern(r) (resp. action(r)) the pattern (resp. the action) of r .

The separator between actions (\bullet) can be understood as a conjunction: the effect of $act_1 \bullet act_2$ is the effect of performing act_1 and act_2 . Intuitively, the atomic action $p(\bar{y}) \mid \phi$ (resp. $\neg p(\bar{y}) \mid \phi$) means that the interpretation of predicate p is made true (resp. false) for instances of free variables in \bar{y} that satisfy ϕ . When ϕ is the true formula \top , we simply write $p(\bar{y})$ instead of $p(\bar{y}) \mid \top$.

A transition rule $f(\bar{x}) : action$ is triggered when a specific event $f(\bar{t})$ occurs: this event is an instance of the rule's pattern by a substitution that must instantiate all parameters by constants.

In order to express easily the relation between two consecutive interpretations (given later in Def. 3), we assume that for any transition rule, for any predicate p , p appears at most once in the event's action (similarly for $\neg p$). This is not a real restriction, because an action of the form $p(\bar{y}_0) \mid \phi \bullet p(\bar{y}_1) \mid \psi$ can be written as $p(\bar{y}) \mid (\bar{y} = \bar{y}_0 \wedge \phi) \vee (\bar{y} = \bar{y}_1 \wedge \psi)$.

As an example, Fig. 2 lists the transitions for the role-based access control system (RBAC).

For later developments, we need to formalise that actions reflect the effect of a transition on a state. We do this by specifying the relation between a given state and its successor with a first-order formula built from the transition rule's actions. This relation consists of three parts: facts set to true, facts set to false, and facts unchanged.

Before giving the relation between consecutive states in details, we define the semantics of these relations. This semantics is adapted from TLA [16]. Given two

$$\begin{aligned}
\text{createSession}(u, s, rs) &: \text{user}(s, u) \bullet \text{role}(s, r_0) \mid r_0 \in rs \\
\text{addActiveRole}(u, s, r) &: \text{role}(s, r) \\
\text{dropActiveRole}(u, s, r) &: \neg \text{role}(s, r) \\
\text{deleteSession}(u, s) &: \neg \text{user}(s, u) \bullet \neg \text{role}(s, r_0) \\
\text{assignRole}(u, r) &: UR(u, r) \\
\text{addInheritance}(r_0, r_1) &: r_0 \leq r_1
\end{aligned}$$

Fig. 2. Transition rules: pairs of a pattern and a list of (*action* | *condition*)

consecutive states \mathcal{I} and \mathcal{I}' , we follow TLA's convention of using non-primed (resp. primed) formulae to indicate formulae interpreted *w.r.t.* \mathcal{I} (resp. \mathcal{I}').

Definition 2 (Two-states relations and their interpretation). *Let \mathcal{I} and \mathcal{I}' be two \mathbb{SP} -interpretations. A two-states relation on \mathcal{I} and \mathcal{I}' is a formula that can contain both non-primed and primed predicates. Non-primed predicates are interpreted by \mathcal{I} while primed predicates are interpreted by \mathcal{I}' . Formally:*

$$\begin{aligned}
\mathcal{I}; \mathcal{I}' \models p(t_1, \dots, t_n) & \quad \text{iff } \mathcal{I}(p)(t_1, \dots, t_n) = \top \\
\mathcal{I}; \mathcal{I}' \models p'(t_1, \dots, t_n) & \quad \text{iff } \mathcal{I}'(p)(t_1, \dots, t_n) = \top \\
\mathcal{I}; \mathcal{I}' \models \neg p(t_1, \dots, t_n) & \quad \text{iff } \mathcal{I}(p)(t_1, \dots, t_n) = \perp \\
\mathcal{I}; \mathcal{I}' \models \neg p'(t_1, \dots, t_n) & \quad \text{iff } \mathcal{I}'(p)(t_1, \dots, t_n) = \perp
\end{aligned}$$

where the t_i 's are variables or constants. Connectors and quantifiers interpretations are similar to the single-state case. As before we leave functions uninterpreted, because our formulae do not include function symbols (even though we use functions to represent events).

We write ϕ' to denote formula ϕ where all predicates have been primed. The formula ϕ' can also be seen as a formula in temporal logic where, instead of priming predicates, they would be enclosed by a next operator **X**.

To simplify the generation of the relation between two consecutive states (see Def. 3), we suppose that actions are put in some normal form. Given a transition rule $f(\bar{x}) : \text{action}$, *action* is normalised iff for any $p(\bar{y}) \mid \phi$ appearing in *action*, we have $\bar{y} \cap \bar{x} = \emptyset$ (similarly for negated actions). Actions are normalised using the following rewrite system:

$$\begin{aligned}
p(\bar{y}_0, x, \bar{y}_1) \mid \phi & \rightarrow p(\bar{y}_0, y, \bar{y}_1) \mid y = x \wedge \phi & \text{where } y \text{ is a fresh variable} \\
\neg p(\bar{y}_0, x, \bar{y}_1) \mid \phi & \rightarrow \neg p(\bar{y}_0, y, \bar{y}_1) \mid y = x \wedge \phi & \text{where } y \text{ is a fresh variable}
\end{aligned}$$

Definition 3 (Relation between consecutive states). *The relation between a \mathbb{SP} -interpretation \mathcal{I} and its successor \mathcal{I}' after a transition r is a first-order formula denoted by $\text{relation}(r)$. built by inspecting the effect of $\text{action}(r)$ on predicates declared in \mathbb{SP} . Let r be " $f(\bar{x}) : \text{act}$ ". For each predicate $p : S_1 \times \dots \times S_m$ declared in \mathbb{SP} , we define:*

$$\text{relation}_p(r) \triangleq \begin{cases} \forall \bar{y}. \phi \Rightarrow p'(\bar{y}) & \text{if } p(\bar{y}) \mid \phi \in \text{act} \\ \top & \text{if } p(\bar{y}) \mid \phi \notin \text{act} \end{cases}$$

$$\begin{aligned}
 \text{relation}_p^{\neg}(r) &\triangleq \begin{cases} \forall \bar{y}. \phi \Rightarrow \neg p'(\bar{y}) & \text{if } \neg p(\bar{y}) \mid \phi \in \text{act} \\ \top & \text{if } \neg p(\bar{y}) \mid \phi \notin \text{act} \end{cases} \\
 \text{relation}_p^{\text{equiv}}(r) &\triangleq \begin{cases} \forall \bar{y}. \neg \phi \wedge \neg \psi \Rightarrow (p'(\bar{y}) \Leftrightarrow p(\bar{y})) & \text{if } p(\bar{y}) \mid \phi \in \text{act} \text{ and } \neg p(\bar{y}) \mid \psi \in \text{act} \\ \forall \bar{y}. \neg \phi \Rightarrow (p'(\bar{y}) \Leftrightarrow p(\bar{y})) & \text{if } p(\bar{y}) \mid \phi \in \text{act} \text{ and } \neg p(\bar{y}) \mid \psi \notin \text{act} \\ \forall \bar{y}. \neg \psi \Rightarrow (p'(\bar{y}) \Leftrightarrow p(\bar{y})) & \text{if } p(\bar{y}) \mid \phi \notin \text{act} \text{ and } \neg p(\bar{y}) \mid \psi \in \text{act} \\ \forall \bar{y}. p'(\bar{y}) \Leftrightarrow p(\bar{y}) & \text{if } p(\bar{y}) \mid \phi \notin \text{act} \text{ and } \neg p(\bar{y}) \mid \psi \notin \text{act} \end{cases}
 \end{aligned}$$

Above, as explained in Def. 2, predicate p' is interpreted as predicate p in interpretation \mathcal{I}' . Finally, the relation between two consecutive states is:

$$\text{relation}(r) \triangleq \exists \bar{x}. \bigwedge_{p \in \mathbb{SP}} \text{relation}_p(r) \wedge \text{relation}_p^{\neg}(r) \wedge \text{relation}_p^{\text{equiv}}(r)$$

For later convenience, we define:

$$\begin{aligned}
 \text{quantifiers}(r) &\triangleq \exists \bar{x} \\
 \text{relation}_{\bar{x}}^{\text{updt}}(r) &\triangleq \bigwedge_{p \in \mathbb{SP}} \text{relation}_p(r) \wedge \text{relation}_p^{\neg}(r) \\
 \text{relation}_{\bar{x}}(r) &\triangleq \bigwedge_{p \in \mathbb{SP}} \text{relation}_p(r) \wedge \text{relation}_p^{\neg}(r) \wedge \text{relation}_p^{\text{equiv}}(r)
 \end{aligned}$$

2.2 Secured Systems

In our framework, transition systems over structures are simply called *systems*. *Security policies* restrict possible transitions to obtain *secured systems*. This section defines these components.

Definition 4 (System). Given a specification \mathbb{SP} , a formula *init* representing initial conditions, and a set τ of transition rules; the system $\mathfrak{S} = [\mathbb{SP}, \text{init}, \tau]$ is the transition system such that:

- the set of initial states is: $\{\mathcal{I}_0 \mid \mathcal{I}_0 \text{ is an } \mathbb{SP}\text{-structure and } \mathcal{I}_0 \models \text{init}\}$.
- the transition relation $\xrightarrow{\text{evt}}^{\mathfrak{S}}$ is such that, for any event $\text{evt} = \sigma(\text{pattern}(r))$ with $r \in \tau$, $\mathcal{I} \xrightarrow{\text{evt}} \mathcal{I}'$ whenever $\mathcal{I}, \mathcal{I}' \models \sigma(\text{relation}(r))$.

Definition 5 (Security policy). Given a specification \mathbb{SP} , a security policy \wp is a function that maps all patterns to \mathbb{SP} -formulae. Given a \mathbb{SP} -interpretation \mathcal{I} , a policy \wp , a transition rule r , and an event $\text{evt} = \sigma(f(\bar{x})) = \sigma(\text{pattern}(r))$, *evt* is authorised by \wp in \mathcal{I} iff $\mathcal{I} \models \sigma(\wp(f(\bar{x})))$ holds.

Fig. 3 shows how to define policy \wp_{RBAC} for the RBAC system given before. Let us suppose that the event $\text{createSession}(\text{Alice}, 0, \{\text{Secretary}, \text{Worker}\})$ occurs in an RBAC system whose state is interpretation \mathcal{I}_{RBAC} given in Fig. 1. The policy checks if the following holds:

$$\mathcal{I}_{RBAC} \models \left(\forall u_0. \neg \text{user}(0, u_0) \wedge \forall r_0. \neg \text{role}(0, r_0) \right) \wedge \forall r_0. r_0 \in \{\text{Secretary}, \text{Worker}\} \Rightarrow \text{UR}(\text{Alice}, r_0)$$

$$\wp_{RBAC} \triangleq \left\{ \begin{array}{l} \text{createSession}(u, s, rs) \mapsto \left(\begin{array}{l} \forall u_0. \neg \text{user}(s, u_0) \wedge \forall r_0. \neg \text{role}(s, r_0) \\ \wedge \forall r_0. r_0 \in rs \Rightarrow UR(u, r_0) \end{array} \right) \\ \text{addActiveRole}(u, s, r) \mapsto \text{user}(s, u) \wedge UR(u, r) \\ \text{dropActiveRole}(u, s, r) \mapsto \text{user}(s, u) \wedge \text{role}(s, r) \\ \text{deleteSession}(u, s) \mapsto \text{user}(s, u) \\ \text{assignRole}(u, r) \mapsto \top \\ \text{addInheritance}(r_0, r_1) \mapsto \neg(r_1 \leq r_0) \end{array} \right\}$$

Fig. 3. Policy for RBAC

But this does not hold, because $\text{user}(0, \text{Bob})$ in \mathcal{I}_{RBAC} contradicts the first conjunct. Having 1 as createSession 's second parameter in the event would make it hold.

Definition 6 (Secured system). Given a system $\mathfrak{S} = [\mathbb{SP}, \text{init}, \tau]$ and a policy \wp , we denote by $\mathfrak{S}_{|\wp} = [\mathbb{SP}, \text{init}, \tau]_{|\wp}$ the \wp -secured system built from \mathfrak{S} . $\mathfrak{S}_{|\wp}$ is the transition system such that:

- the set of initial states is: $\{\mathcal{I}_0 \mid \mathcal{I}_0 \text{ is an } \mathbb{SP}\text{-structure and } \mathcal{I}_0 \models \text{init}\}$
- the transition relation $\xrightarrow[\wp]{\text{evt}}^{\mathfrak{S}}$ is such that, for any event $\text{evt} = \sigma(\text{pattern}(r))$ with $r \in \tau$:

$$\mathcal{I} \xrightarrow[\wp]{\text{evt}}^{\mathfrak{S}} \mathcal{I}' \quad \text{iff} \quad \mathcal{I} \xrightarrow{\text{evt}}^{\mathfrak{S}} \mathcal{I}' \wedge \mathcal{I} \models \sigma(\wp(\text{pattern}(r)))$$

Let $\xrightarrow[\wp]^*_{\mathfrak{S}}$ denote the reflexive transitive closure of this transition relation.

3 Invariants of Secured Systems

Because secured systems are transition systems, they support reasoning about temporal formulae, in particular invariants. Formally, given a secured system $\mathfrak{S}_{|\wp} = [\mathfrak{S}, \text{init}, \tau]_{|\wp}$, formula ϕ is an invariant of $\mathfrak{S}_{|\wp}$ (written $\mathfrak{S}_{|\wp} \models \Box \phi$) if for any initial state \mathcal{I}_0 , $\mathcal{I}_0 \models \phi$ and $\forall \mathcal{I}. \mathcal{I}_0 \xrightarrow[\wp]^*_{\mathfrak{S}} \mathcal{I} \Rightarrow \mathcal{I} \models \phi$. Like in TLA, an invariant can be checked *inductively* i.e. by checking that it is true in the initial state and it is preserved by every step of the transition system. In Definition 3, we wrote the relation that exists between \mathcal{I} and \mathcal{I}' when $\mathcal{I} \xrightarrow{\sigma(\text{pattern}(r))} \mathcal{I}'$: it is $\text{relation}_{\exists}(r)$. Furthermore, the security policy forces $\mathcal{I} \models \sigma(\wp(\text{pattern}(r)))$ to hold. By generalizing over all possible events (i.e. over σ), we have to prove: $\text{init} \Rightarrow \phi$ and for any $r \in \tau$:

$$\phi \wedge [\text{quantifiers}(r). \text{relation}_{\exists}(r) \wedge \wp(\text{pattern}(r))] \Rightarrow \phi' \quad (\text{INV})$$

Using this method, we can prove that the RBAC system shown before, secured by policy \wp_{RBAC} , enforces the invariant that sessions are either unassigned or assigned to a *unique* user:

$$\forall s. (\forall u. \neg \text{user}(s, u)) \vee (\exists! u. \text{user}(s, u))$$

For example, to prove that Fig. 2's transition rule *createSession* preserves this invariant, we have to prove:

$$\begin{array}{l}
\forall s_0. (\forall u_0. \neg user(s_0, u_0)) \vee (\exists! u_0. user(s_0, u_0)) \\
\wedge \exists u \ s \ rs. \\
\quad user'(s, u) \wedge \forall r_0. r_0 \in rs \Rightarrow role'(s, r_0) \\
\quad \wedge \forall s_0 \ u_0. s_0 \neq s \vee u_0 \neq u \Rightarrow (user'(s_0, u_0) \Leftrightarrow user(s_0, u_0)) \\
\quad \wedge \forall s_0 \ r_0. s_0 \neq s \vee r_0 \notin rs \Rightarrow (role'(s_0, r_0) \Leftrightarrow role(s_0, r_0)) \\
\quad \wedge \forall u_0 \ r_0. UR'(u_0, r_0) \Leftrightarrow UR(u_0, r_0) \\
\quad \wedge \left(\begin{array}{l} \forall u_0. \neg user(s, u_0) \wedge \forall r_0. \neg role(s, r_0) \\ \wedge \forall r_0. r_0 \in rs \Rightarrow UR(u, r_0) \end{array} \right) \\
\quad \Downarrow \\
\forall s_0. (\forall u_0. \neg user'(s_0, u_0)) \vee (\exists! u_0. user'(s_0, u_0))
\end{array}
\begin{array}{l}
\} \phi \\
\} quantifiers(r) \\
\} relation_{\bar{x}}(r) \\
\} \wp(pattern(r)) \\
\} \phi'
\end{array}$$

This implication can be proven with any theorem prover that accepts first-order logic with equality and set theory. Alternatively, the RBAC system and its security policy can be encoded in TLA [16] or B [3], and this invariant can be proven using these tools.

4 Abduction algorithm

In this section, we present our abduction algorithm. In Sec. 4.1, we show a standard abduction algorithm for first-order logic and sequent calculus based on Mayer and Pirri [19]. In Sec. 4.2, we show how our application domain helps in selecting *good* candidate security policies³ for abduction, then in Sec. 4.3, we show how it helps generating additional candidates.

We assume here that we have a proof search procedure in first-order logic. Given $\mathbb{SP} = (\Sigma, \mathbb{T})$, we write $\phi \vdash_{\mathbb{SP}} \psi$ to denote that ϕ and the theory \mathbb{T} entail ψ . We omit the \mathbb{SP} subscript when it is clear from the context. We require the proof search procedure to be correct *w.r.t.* \models , *i.e.*:

$$\text{if } \phi \vdash_{(\Sigma, \mathbb{T})} \psi \text{ then } \mathbb{T} \models \phi \Rightarrow \psi$$

4.1 Abduction Problem

An abduction problem in our application domain is a triple $\langle \mathbb{SP}; \tau; \phi \rangle$ where ϕ is an invariant that should be respected by any system $\mathfrak{S} = [\mathbb{SP}, init, \tau]$. A solution to this abduction problem is a policy \wp that associates to each transition rule $r \in \tau$ a \mathbb{SP} -formula such that transitions of any secured system $\mathfrak{S}_{|\wp}$ preserves ϕ . This means that for any $r \in \tau$, \wp must satisfy:

$$\phi \wedge (quantifiers(r).relation_{\bar{x}}(r) \wedge \wp(pattern(r))) \Rightarrow \phi'$$

³ Abduction is mainly used in the AI community, where the term *explanation* denotes abducted formulae. In this paper, however, we prefer the term “candidate policy” or “candidate”.

Following Mayer and Pirri [19], we use unfinished proofs in classical natural deduction (specifically **NK** [13]) to find an appropriate $\wp(\text{pattern}(r))$ above. A *sequent* ς in natural deduction is a construct of the form $\psi_1, \dots, \psi_n \vdash \xi$ where ψ_1, \dots, ψ_n are *hypotheses* and ξ is the *goal*. In other terms, such a sequent can be interpreted as the implication $\psi_1 \wedge \dots \wedge \psi_n \Rightarrow \xi$. With Γ ranging over sets of formulae, we sometimes write sequents $\Gamma \vdash \xi$.

Given a sequent, proof trees are built using inference rules [13]. In a proof tree, a leaf is *closed* if it is an axiom. Otherwise it is *open*. We say that a formula ψ is the candidate policy of an open leaf ς , if adding ψ to ς 's hypotheses closes ς . Usually [19], a candidate policy of an open leaf ς is ς 's goal or the negation of an hypothesis of ς . This definition of candidate policy, however, is too coarse for our application domain. That is why we add *contexts* in order to complete candidate policies with the part of the hypotheses involving non-parameter variables:

Definition 7 (Context of a formula). *Given a transition rule r , a sequent $\psi_1, \dots, \psi_n \vdash \xi$, and a formula φ ; the context of φ in $\psi_1, \dots, \psi_n \vdash \xi$ (written $\text{ctx}(r, \{\psi_1, \dots, \psi_n\}, \varphi)$) is a formula of the form $Q_1 x_1 \dots Q_n x_n. \zeta \Rightarrow \varphi$ such that (1) ζ is a conjunction of the formulae ψ_i ($1 \leq i \leq n$) that contain φ 's variables that are not parameters of rule r and (2) $Q_1 x_1, \dots, Q_n x_n$ quantify over those variables.*

For example, given $r = \text{createSession}(u, s, rs)$, we have:

$$\begin{aligned} \text{ctx}(r, \{u_0 \neq u, u_0 \neq \text{Bob}\}, \text{user}(s, u)) &\triangleq \text{user}(s, u) \\ \text{ctx}(r, \{u_0 \neq u, u = \text{Bob}\}, \text{user}(s, u_0)) &\triangleq \forall u_0. u_0 \neq u \Rightarrow \text{user}(s, u_0) \\ \text{ctx}(r, \{u_0 \neq u, s_0 = s\}, \text{user}(s_0, u_0)) &\triangleq \forall s_0 u_0. u_0 \neq u \wedge s_0 = s \Rightarrow \text{user}(s_0, u_0) \end{aligned}$$

Computing contexts consists in (1) gathering hypotheses about non-parameter variables in candidate policies (variables that are not quantified and that are not parameters of the event) and (2) quantifying appropriately these variables. Point (2) is similar to *reverse skolemization* [9, 19]. Because later use of contexts is no more complex than the above examples and because computing contexts is orthogonal to this paper's topic, we do not detail it. Computing contexts is useful to return weaker security policies in the following sense: for any r, Γ and φ , if $\text{ctx}(r, \Gamma, \varphi) = Q_1 x_1, \dots, Q_n x_n. \zeta \Rightarrow \varphi$, we have $\varphi \Rightarrow (\zeta \Rightarrow \varphi)$.

Candidate policies are built in two ways:

Definition 8 (Positive candidate policy). *Given an abduction problem $\langle \mathbb{S}\mathbb{P}; r; \phi \rangle$, the positive candidate policy of the open leaf $\Gamma \vdash \xi$ is $\text{ctx}(r, \Gamma, \xi) \Rightarrow \xi$.*

Definition 9 (Negative candidate policies). *Given an abduction problem $\langle \mathbb{S}\mathbb{P}; r; \phi \rangle$, the set of negative candidate policies of the leaf $\Gamma \vdash \xi$ is:*

$$\{\varphi \mid \varphi = (\text{ctx}(r, \Gamma, \psi) \Rightarrow \neg\psi) \text{ and } \psi \in \Gamma\}$$

Candidate policies are used to build solutions to abduction problems:

Definition 10 (Solution to an abduction problem). *Given an abduction problem $\langle \mathbb{S}\mathbb{P}; r; \phi \rangle$, and a proof tree of $\phi \wedge (\text{quantifiers}(r). \text{relation}_{\vec{x}}(r) \wedge \wp(r)) \vdash \phi'$ whose open leaves are $\varsigma_1, \dots, \varsigma_n$, a solution to the abduction problem consists of:*

1. n non-empty sets of formulae $\Gamma_1, \dots, \Gamma_n$ such that for any $1 \leq i \leq n$, Γ_i is the set of (positive and negative) candidate policies of leaf ς_i .
2. A choice function ϵ such that $\epsilon(\Gamma_i) \in \Gamma_i$.

Given those two elements, a policy preserving invariant ϕ when transition r fires in a system $[\mathbb{SP}, \text{init}, r]$ is $\wp = \{\text{pattern}(r) \mapsto \epsilon(\Gamma_1) \wedge \dots \wedge \epsilon(\Gamma_n)\}$. This definition generalises straightforwardly to systems with multiple transition rules.

When abducting policies for systems with multiple transitions, we can obtain contradictory policies (such as policy ψ for an event and policy $\neg\psi$ for another event). While we provide no way - in this paper - to detect such inconsistent policies (because our approach is *per* transition), this could be detected by additional semantical checks involving candidate policies abducted for different events. This problem does not harm, however, the soundness of our approach.

Theorem 1 (Correctness of abduction). *Given a system $\mathfrak{S} = [\mathbb{SP}, \text{init}, \tau]$, a solution \wp to the abduction problem $\langle \mathbb{SP}; \tau; \phi \rangle$ and a proof that \mathfrak{S} 's initial states satisfy invariant ϕ (i.e. $\mathbb{SP} \models \text{init} \Rightarrow \phi$); then $\mathfrak{S}_{|\wp} \models \Box \phi$.*

While Def. 10 shows how to build a policy preserving an invariant, it can be improved in two ways. First, some candidate policies are not acceptable in our framework and must be filtered out. Second, for abduction to succeed all sets of candidate policies $\Gamma_1, \dots, \Gamma_n$ must be *non-empty*. If the brute force method does not allow to close all branches, more candidates have to be found. Sec. 4.2 and Sec. 4.3 tackle these two problems.

4.2 Domain-specific filtering of candidate policies

Because abduction is all about finding *good* candidate policies, we now define various *filtering* criteria that exclude “bad” candidates.

The first level of filtering eliminates candidate policies syntactically. Syntactical filtering rules out policies that are unrelated to the transition considered or that constrain the next state. E.g, filtering forbids invariants to be candidate policies, because invariants do not have free variables.

Definition 11 (Acceptable candidate policy). *Formula ψ is an acceptable candidate policy for the abduction problem $\langle \mathbb{SP}; r; \phi \rangle$ iff all the following conditions hold:*

1. *At least one parameter of rule r occurs in ψ as a free variable.*
2. *ψ contains a user-defined predicate.*
3. *ψ contains no primed predicate.*

The second level of filtering eliminates candidate policies semantically, because it requires proof search. Intuitively, we should check that a candidate policy ψ is *not* a consequence of the transition’s action. Otherwise, having this candidate as a policy would mean “require ψ to be true before setting ψ to true”.

Definition 12 (Good candidate policy). *Formula ψ is a good candidate policy for the abduction problem $\langle \text{SP}; r; \phi \rangle$ iff the following formula holds:*

$$\neg(\text{quantifiers}(r).\text{relation}_{\neq}^{\text{updt}}(r) \Rightarrow \psi')$$

The two levels of filtering eliminate some candidate policies in Def. 10. Instead of considering all candidates of open leaves, we consider only good candidates. Yet, it is possible to get several good candidates so that some choice must be made. In this case, we order candidate policies to help make this choice.

Definition 13 (Order on candidate policies). *A choice strategy is a function f whose domain is the set of formulae and whose range is some set \mathcal{R} such that there is a relation $\leq_{\mathcal{R}}$ such that $(\mathcal{R}, \leq_{\mathcal{R}})$ is a partial order. We say that ψ is a better candidate policy than ϕ if $f(\phi) \leq f(\psi)$.*

A standard order (called minimality [19, Sec. 1.1]) is the function f such that $f(\phi) \leq f(\psi)$ iff $\phi \vdash \psi$. While this order is powerful, it is only partial (there are ϕ and ψ such that $\phi \not\vdash \psi$ and $\psi \not\vdash \phi$). Due to our application domain, however, we can provide other orders. For example, cheap total orders include the functions that count the number of free or quantified variables in candidate policies. Intuitively, such orders are useful because - in our application domain - (1) Free variables in candidate policies are parameters of transition rules (e.g. u , s , and rs in *createSession*'s policy in Fig. 3). Because policies typically constrain parameters of transition rules, best candidate policies maximise the number of free variables. (2) Universally quantified variables occur as arguments of predicates (e.g. u_0 and r_0 in *createSession*'s policy in Fig. 3). Because this often indicates too strong candidate policies, best candidates minimise the number of universally quantified variables. As usual, such orders can be combined. For example, let f (resp. g) be the function of type *Formula* $\rightarrow \mathbb{N}$ that counts free (resp. universally quantified) variables in formulae. A cheap and useful order is the syntactical lexicographical order \leq_{vars} built from f and g :

$$\phi \leq_{\text{vars}} \psi \quad \text{iff} \quad f(\phi) < f(\psi) \text{ or } (f(\phi) = f(\psi) \text{ and } g(\phi) > g(\psi))$$

Orders on candidate policies may already arise in Def. 10, because the choice function ε can be defined in terms of orders. Typically, ε must satisfy: $\forall \Gamma. \forall \psi \in \Gamma. \varepsilon(\Gamma) \leq \psi$.

4.3 Domain-specific generation of candidate policies

As Def. 10 shows, abduction fails if *one* open leaf does not have any candidate policy. In this section, we show how to avoid some failures of abduction by generating more candidates than the standard techniques given in Def. 8 (positive candidates) and Def. 9 (negative candidates).

First, we show how our application domain allows the generation of additional negative candidate policies.

Definition 14 (Additional negative candidate policies). *Given an abduction problem $\langle \mathbb{S}\mathbb{P}; r; \phi \rangle$ and an open leaf $\Gamma \vdash \xi$, the set of negative candidate policies of this leaf is:*

$$\left\{ \varphi \mid \begin{array}{l} \varphi = (ctx(r, \Gamma, \psi) \Rightarrow \neg\psi), \zeta \in \Gamma, \Gamma \Rightarrow \psi \text{ holds,} \\ \text{and } \psi \text{ is } \zeta \text{ where primes have been removed} \end{array} \right\}$$

Def. 14 tries to “guess” negative candidate policies by removing primes from hypotheses of open leaves. This guessing is motivated by three reasons specific to our application domain. First, hypotheses of sequents often contain primed predicates. This stems from the fact that invariants often have the form $Q_1 x_1 \dots Q_n x_n. \psi' \Rightarrow \xi'$ where Q_1, \dots, Q_n are quantifiers and ψ' and ξ' are quantifier free. Because proofs of such formulae typically follow the pattern:

$$\frac{\frac{\psi' \vdash \xi'}{\vdash \psi' \Rightarrow \xi'} \Rightarrow\text{-R}}{\vdash Q_1 x_1 \dots Q_n x_n. \psi' \Rightarrow \xi'} \text{ (Quantifier elimination)}$$

primed formulae (*e.g.* ψ') end up in hypotheses. The second reason that motivates Def. 14 is that candidate policies should contain no primes. The third reason is that formulae obtained after removing primes can often be proved, because - by construction - proof obligations contain equivalences between non-primed and primed predicates (see Def. 3) as hypotheses.

Second, we show how to generate additional positive candidate policies. To do this, we impose the proof search algorithm to follow a general pattern, which is guided by our application domain. First, let us recall that proof obligations in our framework have exactly this form:

$$\phi \wedge (\text{quantifiers}(r).relation_{\exists}(r) \wedge \psi) \Rightarrow \phi'$$

Now, suppose that the invariant ϕ is in prenex form *i.e.* it has the form $Q_1 x_1, \dots, Q_n x_n. \xi$ where ξ is quantifier free and $Q_1, \dots, Q_n \in \{\forall, \exists\}$. This is not a restriction, because any first-order formula can be transformed into an equivalent prenex formula. Furthermore, we know that $\text{quantifiers}(r)$ is a sequence of existential quantifiers (see Def. 3). Therefore, we impose the proof search to start as follows: (1) remove the top-level implication, (2) remove all quantifiers in front of ϕ' , (3) split the conjunction in the hypotheses, (4) remove all quantifiers in front of $\text{quantifiers}(r).relation_{\exists}(r) \wedge \psi$, and (5) perform case splits on equality and membership between variables introduced in steps 2 and 4.

To help the reader understand this proof strategy, we illustrate it for a transition rule with a single parameter y and the invariant $\forall x, \bar{x}. \phi$. We abbreviate $relation_{\exists}(r) \wedge \psi$ by ξ :

$$\frac{y = x, y \notin \bar{x}, \phi, \xi \vdash \phi' \quad y = x, y \in \bar{x}, \phi, \xi \vdash \phi' \quad y \neq x, y \notin \bar{x}, \phi, \xi \vdash \phi' \quad y \neq x, y \in \bar{x}, \phi, \xi \vdash \phi'}{\frac{\frac{\frac{\frac{\phi, \xi \vdash \phi'}{\phi, \exists y. \xi \vdash \phi'} \text{ Step (4)}}{\phi \wedge \exists y. \xi \vdash \phi'} \text{ Step (3)}}{\phi \wedge \exists y. \xi \vdash \phi'} \text{ Step (2)}}{\vdash \phi \wedge \exists y. \xi \vdash \forall x, \bar{x}. \phi'} \text{ Step (1)}} \text{ Step (5)}$$

Why is this proof search pattern relevant for our application domain ? To answer this question, suppose we want to enforce invariant $\forall x.p(x)$. In events that modify the truth value of $p(y)$ for some y , the proof of the invariant will perform a case split: (1) show that $p(x)$ holds for any $x \neq y$ and (2) show that $p(y)$ holds. Typically, goal (1) follows from the hypotheses that the invariant holds before the event (because the truth value of $p(x)$ is unchanged in the transition); while goal (2) follows from the policy (because the truth value of $p(y)$ is changed by the transition). Doing case split in low positions in proof trees impacts abduction in two ways: first, it helps generating good candidate policies, because the generated proof trees distinguish between parameters of the transition rule from other variables; this mirrors the fact that policies constrain differently parameters of transition rules and other variables (as all rules of Fig. 3's RBAC policy exemplify). Second, having case splits in proof trees permits to gather positive candidate policies in lower positions than open leaves, as we detail in the next paragraphs.

To justify how to find additional positive candidate policies, let us consider rule \Rightarrow -E of the sequent calculus:

$$\frac{\psi_1, \dots, \psi_n \vdash \xi \Rightarrow \varphi \quad \psi_1, \dots, \psi_n \vdash \xi}{\psi_1, \dots, \psi_n \vdash \varphi} \Rightarrow\text{-E}$$

Rule \Rightarrow -E means: to prove φ when $\xi \Rightarrow \varphi$ holds, it suffices to prove ξ . In this inference rule, φ can be seen as the *reason* why ξ appears as a goal.

Definition 15 (Additional positive candidate policy). *Given an abduction problem $\langle \text{SP}; r; \phi \rangle$ and an open leaf $\Gamma \vdash \xi$, the corresponding positive candidate policy is obtained as follows:*

- *If $\text{ctx}(r, \Gamma, \xi) \Rightarrow \xi$ is a good candidate, it is the positive candidate policy.*
- *Otherwise, the positive candidate policy is the goal of the first sequent (below the open leaf and above case splits involving parameters from the transition rule) whose inference rule is \Rightarrow -E and whose goal is a good candidate policy.*

Def. 15 means that if the goal of the open leaf is inappropriate, the algorithm looks for the reason why this branch of the proof tree has been started during proof search and returns this reason as the positive candidate policy.

Collecting positive candidate policies by traversing proof trees from top to bottom can be inadequate if it goes all way down to the root of the proof tree (*i.e.* the root's goal is taken as the leaf's positive candidate policy). Because we impose a proof strategy, however, we can stop traversing the proof tree from top to bottom when a case split involving parameters of the transition rule is encountered. This avoids returning the desired invariant as candidate policy. Furthermore, because proof trees occurring above case splits represent radically different cases, candidate policies for a given open leaf should not be chosen below a case split: such a candidate would not be specific to the open leaf's case.

5 Case study: Role-based access control systems

We show how to synthesise a policy for event $createSession(u, s, \bar{r})$ that enforces the invariant that activated roles are assigned to users:

$$\forall s u r. user(s, u) \wedge role(s, r) \Rightarrow UR(u, r)$$

Given the effect of event $createSession(u, s, \bar{r})$ (it is $user(s, u) \bullet role(s_0, r_0) \mid s_0 = s \wedge r_0 \in \bar{r}$ as shown in Fig. 2), abduction consists in finding ψ such that:

$$\begin{array}{l} \forall s_0 u_0 r_0. user(s_0, u_0) \wedge role(s_0, r_0) \Rightarrow UR(u_0, r_0) \\ \wedge \exists u s \bar{r}. \\ \quad user'(s, u) \wedge \forall r_0. r_0 \in \bar{r} \Rightarrow role'(s, r_0) \\ \quad \wedge \forall s_0 u_0. s_0 \neq s \vee u_0 \neq u \Rightarrow (user'(s_0, u_0) \Leftrightarrow user(s_0, u_0)) \\ \quad \wedge \forall s_0 r_0. s_0 \neq s \vee r_0 \notin \bar{r} \Rightarrow (role'(s_0, r_0) \Leftrightarrow role(s_0, r_0)) \\ \quad \wedge \forall u_0 r_0. UR'(u_0, r_0) \Leftrightarrow UR(u_0, r_0) \\ \quad \wedge \psi \end{array} \left. \vphantom{\begin{array}{l} \forall s_0 u_0 r_0. user(s_0, u_0) \wedge role(s_0, r_0) \Rightarrow UR(u_0, r_0) \\ \wedge \exists u s \bar{r}. \\ \quad user'(s, u) \wedge \forall r_0. r_0 \in \bar{r} \Rightarrow role'(s, r_0) \\ \quad \wedge \forall s_0 u_0. s_0 \neq s \vee u_0 \neq u \Rightarrow (user'(s_0, u_0) \Leftrightarrow user(s_0, u_0)) \\ \quad \wedge \forall s_0 r_0. s_0 \neq s \vee r_0 \notin \bar{r} \Rightarrow (role'(s_0, r_0) \Leftrightarrow role(s_0, r_0)) \\ \quad \wedge \forall u_0 r_0. UR'(u_0, r_0) \Leftrightarrow UR(u_0, r_0) \\ \quad \wedge \psi \end{array}} \right\} \phi$$

$$\Downarrow$$

$$\forall s_0 u_0 r_0. user'(s_0, u_0) \wedge role'(s_0, r_0) \Rightarrow UR'(u_0, r_0) \left. \vphantom{\forall s_0 u_0 r_0. user'(s_0, u_0) \wedge role'(s_0, r_0) \Rightarrow UR'(u_0, r_0)} \right\} \phi'$$

We detail a classical proof trial of the formula above (see this trial in Coq [1]) that follows all steps of the proof search strategy defined in 4.3. In this example, this strategy transforms the goal into 8 subgoals:

- (1) $hypos \wedge s_0 = s \wedge u_0 = u \wedge r_0 \in \bar{r} \vdash user'(s, u) \wedge role'(s, r_0) \Rightarrow UR'(u, r_0)$
- (2) $hypos \wedge s_0 = s \wedge u_0 = u \wedge r_0 \notin \bar{r} \vdash user'(s, u) \wedge role'(s, r_0) \Rightarrow UR'(u, r_0)$
- ⋮
- (8) $hypos \wedge s_0 \neq s \wedge u_0 \neq u \wedge r_0 \notin \bar{r} \vdash user'(s_0, u_0) \wedge role'(s_0, r_0) \Rightarrow UR'(u_0, r_0)$

We now show how a typical proof trial for subgoal (1) proceeds. In our application domain, a standard proof search strategy is to prove goals containing primed predicates by transforming them into non-primed predicates. This is reasonable, because we have equivalences between primed and non-primed predicates in hypotheses. This works, because our hypotheses (the invariant and policies) are not-primed. This proof search strategy proceeds as follows on subgoal (1) (where we omit some trivial leafs and where ξ abbreviates $s_0 = s \wedge u_0 = u \wedge r_0 \in \bar{r}$):

$$\frac{\frac{\frac{\text{stuck!}}{\phi, hypos, \xi, user'(s, u), role'(s, r_0) \vdash \boxed{user(s, u)}}{\text{stuck!}}}{\dots \vdash role(s, r_0)}}{\phi, hypos, \xi, user'(s, u), role'(s, r_0) \vdash user(s, u) \wedge role(s, r_0)} \wedge\text{-I}}{\frac{\phi, hypos, \xi, user'(s, u), role'(s, r_0) \vdash \boxed{UR(u, r_0)}}{\phi, hypos, \xi, user'(s, u), role'(s, r_0) \vdash UR'(u, r_0)} \Rightarrow\text{-E and } hypos[5]}}{\phi, hypos, \xi \vdash user'(s, u) \wedge role'(s, r_0) \Rightarrow UR'(u, r_0)} \Rightarrow\text{-I}$$

Proof search is stuck in two leafs, because each respective goal is a non-primed predicate, and no hypothesis matches it. We detail how the candidate policies for the left open leaf are obtained (the case of the other leaf is similar). This leaf's

goal ($user(s, u)$) is an acceptable positive candidate: it contains only non-primed user-defined predicates and two variables of the event (s and u). It is *not*, however, a good candidate, because it is incompatible with $createSession$'s actions: $createSession$ sets $user(s, u)$ to true. Incompatibility is detected by proving the following formula (where moustaches refers to Def. 12 to show how the sequent is built systematically):

$$\neg \underbrace{\exists u \ s \ \bar{r}.}_{\text{quantifiers}(createSession(u, s, \bar{r}))} \underbrace{user'(s, u) \wedge \forall r_0. r_0 \in \bar{r} \Rightarrow role'(s, r_0)}_{\text{relation}_{\frac{updt}{\bar{r}}}(createSession(u, s, \bar{r}))} \Rightarrow \underbrace{user'(s, u)}_{\psi'}$$

Because the candidate policy $user(s, u)$ is not a good one, we use Def. 15 to look for other candidates in lower positions in the proof tree (as indicated by grey background). Because the positive candidate policy $\forall r_0. r_0 \in \bar{r} \Rightarrow UR(u, r_0)$ (built from the goal $UR(u, r_0)$ and the goal's context) (1) is under an application of rule \Rightarrow -E and (2) is a good candidate, it is returned by the algorithm.

This closes subgoal (1). However, the proof is stuck in subgoal (2) in the same position as subgoal (1): the resulting proof tree is the same except that ξ abbreviates $s_0 = s \wedge u_0 = u \wedge r_0 \notin \bar{r}$. In this subgoal, the positive candidate policy $\forall r_0. r_0 \notin \bar{r} \Rightarrow UR(u, r_0)$ is produced. In addition, negative candidates are found. Because $user'(s, u)$ and $role'(s, r_0)$ occur in the leaf's hypotheses; the provability of $user(s, u)$ and $role(s, r_0)$ are checked (as in Def. 14). Because $role(s, r_0)$ is provable and is good, the negative candidate policy $\forall r_0. r_0 \notin \bar{r} \Rightarrow \neg role(s, r_0)$ is produced. To recap, after abducting the constraint $\forall r_0. r_0 \in \bar{r} \Rightarrow UR(u, r_0)$ for goal (1), one of the two candidate policies below should be chosen for goal (2):

$$\forall r_0. r_0 \notin \bar{r} \Rightarrow UR(u, r_0) \quad \text{or} \quad \forall r_0. r_0 \notin \bar{r} \Rightarrow \neg role(s, r_0)$$

Because choosing the first candidate policy would make the constraint equivalent to $\forall r_0. UR(u, r_0)$ (which is very restrictive), it indicates that the second candidate should be chosen. Sec. 4's syntactical order \leq_{vars} selects the good candidate if we use the already abducting constraint, because the following holds:

$$\left(\begin{array}{l} \forall r_0. r_0 \in \bar{r} \Rightarrow UR(u, r_0) \\ \wedge \forall r_0. r_0 \notin \bar{r} \Rightarrow UR(u, r_0) \end{array} \right) \leq_{vars} \left(\begin{array}{l} \forall r_0. r_0 \in \bar{r} \Rightarrow UR(u, r_0) \\ \wedge \forall r_0. r_0 \notin \bar{r} \Rightarrow role(s, r_0) \end{array} \right)$$

Choosing the second candidate policy and completing the proof leads to abduct two candidates: $\forall u_0 \ r_0. u_0 \neq u \wedge r_0 \in \bar{r} \Rightarrow UR(u_0, r_0)$ and $\forall u_0. u_0 \neq u \Rightarrow \neg user(s, u_0)$. The second candidate (which is the "good" one) is automatically selected by the order \leq_{vars} . This third abduction step suffices to finish the proof. Finally, the abducting constraint for $createSession$ is the formula ψ below while the formula ξ is the constraint given in Fig. 3:

$$\psi \triangleq \begin{array}{l} \forall r_0. r_0 \in \bar{r} \Rightarrow UR(u, r_0) \\ \wedge \forall r_0. r_0 \notin \bar{r} \Rightarrow \neg role(s, r_0) \\ \wedge \forall u_0. u_0 \neq u \Rightarrow \neg user(s, u_0) \end{array} \quad \xi \triangleq \begin{array}{l} \forall r_0. r_0 \in \bar{r} \Rightarrow UR(u, r_0) \\ \wedge \forall r_0. \neg role(s, r_0) \\ \wedge \forall u_0. \neg user(s, u_0) \end{array}$$

While the two constraints are close, they are not equivalent. The ξ constraint entails the abducting constraint ψ (hence the abducting constraint is less restrictive). The difference is that the abducting constraint allows a user u to open

twice session s : ψ does not imply $\neg user(s, u) \wedge \forall r_0. \neg role(s, r_0)$. On the contrary ξ implies $\neg user(s, u) \wedge \forall r_0. \neg role(s, r_0)$ (*i.e.* ξ requires u not to have opened s before). The difference between the abducted constraint and the human-written constraint, however, is not a defect of our abduction algorithm. The difference stems from the invariant used to guide abduction: this invariant does not forbid a user to open a session multiple times. We cannot, however, express the invariant that users are forbidden to open a session multiple times, because it is a two states properties. To support such invariants, we need temporal reasoning (at least the \mathbf{X} operator).

This Section's examples as well as middle-size additional examples are available online [1]. These experiments validate our choice of heuristics.

6 Related Work and Conclusion

Related work. Among a rich literature on security policies (see for instance [10] for policy specification languages), our specification framework is in the line of logic-based languages providing a well-understood formalism, amenable to verification, proof and analysis. Close frameworks based on Datalog are [11] and [5]. In [11] security properties are analysed using model checking formulae in first-order temporal logic and in [5], sequences of user actions are analysed with a proof system.

Concerning invariant verification, the closest related work is Lamport's TLA [16, 20]. First, like in TLA, the effects of transitions (*i.e.* actions) are expressed with first-order formulae. Second, our rules for proving invariants (in Sec. 3) are similar to TLA's rules. But contrary to TLA, we do not allow users to write arbitrary first-order formula as actions: users have to write actions using $p(\bar{y}) \mid \phi$ and $\neg q(\bar{z}) \mid \psi$ predicates. This permits to have well-structured relations between two consecutive states. Such well-structured relations simplify the presentation of our abduction algorithm (see *e.g.* the use of $relation_{\bar{x}}^{updt}$ in Def. 12). Furthermore, our language of actions induce relations in first-order logic with equality and set theory, while TLA's relations contain non-standard operators (*e.g.* [17, p. 3]) to express changes to the interpretation of predicates. Another difference is that, in TLA, security policies would be specified as part of actions. Consequently, in TLA, plug-in different security policies and comparing them (for a single system) is not as easy as in our approach.

Concerning abduction, the closest related work is [19]'s abduction algorithm for first-order logic. Like Mayer and Pirri, our abduction algorithm uses proof trees in sequent calculi. However, while Mayer and Pirri's algorithm gathers positive justifications by inspecting solely opened leafs, our algorithm has a top to bottom strategy: it first gathers positive justifications in opened leafs; but it can also recursively inspect goals in lower positions in the proof tree. Another crucial difference is that our application domain allows us to define criteria (Defs. 11 and 12) to filter out undesirable explanations and to classify possible explanations.

The literature on abduction for logic programs is consequent (see [14, 23, 4, 6, 25] and [15] for a survey). These works use techniques that do not fit our setting of first-order logic and sequent calculi. Abduction for logic programs has good termination and completeness result which cannot be achieved in first-order logic. Russo *et al.* [23] use abductive logic programming to detect violation of invariants in event-based systems. Contrary to us, they use abduction in “refutation mode”, where abducted formulae represent possible violations of invariants which can be used to modify either the system or the faulty invariant. To our knowledge, the only other work on abduction for security policies is by Becker *et al.* [6] who use abduction for various applications: to write and debug policies, to explain to users when access is denied, and to compute missing credentials automatically. While the first application is close to ours; Becker *et al.* do not use invariants to guide abduction and they do not tune abduction with domain-specific hypotheses like we do.

Further work and Conclusion. First, we presented a framework to specify transition systems restricted by security policies, based on first-order logic. The framework is modular in the sense that the system and the policy that restricts its transitions are given separately, but using a common language. This gives the possibility to plug-in different security policies and to compare them.

Second, we proposed an abduction algorithm that infers security policies by using invariants as guides. These invariants are expressing security properties. We showed how the specification formalism of transition systems restricted by policies is appropriate to policy synthesis. From the logical point of view, the specificities of this application domain naturally induces both syntactic heuristics and a proof search methodology that allowed us to tune abduction. We have shown how to filter out explanations not suitable as security policies and how to generate additional candidates for possible security policies.

Contrary to most previous work on abduction, our approach is based on first-order logic instead of logic programming. Indeed decidability and complexity results are harder to obtain, but first-order logic is standard to specify systems, used for instance in TLA [16] or B [3]. While we do not want to stick to a specific framework, our algorithm could be adapted without significant effort to infer part-of TLA’s actions and to infer part-of B’s preconditions.

Future work includes evaluating the quality of our heuristics. This is complex, because these heuristics depend on the proof search strategy, on the way proof trees are inspected, and on the choice criteria between candidate explanations. For that, we plan to implement the abduction algorithm in existing theorem provers. This is handy in interactive theorem provers such as Coq [2] or Isabelle [21] that provide rich interfaces to inspect the proof’s state. The algorithm can also be implemented in automatic theorem provers for first-order logic based on tableau or sequent calculi [22, 7].

Another advantage of the synthesis approach is its incremental feature. In this paper, for clarity, we used only one invariant to guide abduction. In practice, however, systems often have to ensure multiple, often unrelated invariants. Our abduction algorithm can be easily made incremental to allow multiple invariants.

Acknowledgements. We are very grateful to Émilie Balland and Paul Brauner, both for their careful reading of this paper and their numerous suggestions that helped improving our work. We also thank Katsumi Inoue for his detailed answers to our technical questions.

References

1. Examples of abduction with Coq [2]: <http://www-sop.inria.fr/everest/Clement.Hurlin/abduction.tgz>
2. The Coq proof assistant: <http://coq.inria.fr>
3. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
4. Alrajeh, D., Ray, O., Russo, A., Uchitel, S.: Using abduction and induction for operational requirements elaboration. *Journal of Applied Logic* 7(3), 275–288 (2009), special issue: Abduction and Induction in Artificial Intelligence
5. Becker, M.Y., Nanz, S.: A logic for state-modifying authorization policies. In: European Symposium On Research In Computer Security. *Lecture Notes in Computer Science*, vol. 4734, pp. 203–218. Springer-Verlag (2007)
6. Becker, M.Y., Nanz, S.: The role of abduction in declarative authorization policies. In: Symposium on Practical Aspects of Declarative Languages. *Lecture Notes in Computer Science*, vol. 4902, pp. 84–99 (2008)
7. Bonichon, R., Delahaye, D., Doligez, D.: Zenon : An extensible automated theorem prover producing checkable proofs. In: Dershowitz, N., Voronkov, A. (eds.) *Logic for Programming Artificial Intelligence and Reasoning*. vol. 4790, pp. 151–165. Springer-Verlag (2007)
8. Bourdier, T., Cirstea, H., Jaume, M., Kirchner, H.: Formal Specification and Validation of Security Policies (2010), submitted to the Brazilian Symposium on Formal Methods.
9. Cox, P.T., Pietrzykowski, T.: A complete, nonredundant algorithm for reversed skolemization. In: Bibel, W., Kowalski, R.A. (eds.) *Conference on Automated Deduction*. *Lecture Notes in Computer Science*, vol. 87, pp. 374–385. Springer-Verlag (1980)
10. Damianou, N.C., Bandara, A.K., Sloman, M.S., Lupu, E.C.: A survey of policy specification approaches. Tech. rep., Imperial College of Science Technology and Medicine (2002)
11. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Specifying and reasoning about dynamic access-control policies. In: *International Joint Conference on Automated Reasoning*. *Lecture Notes in Computer Science*, vol. 4130, pp. 632–646 (2006)
12. Ferraiolo, D., Kuhn, D.R., Chandramouli, R.: Role-based access control. *Computer security series*, Artech House (2003)
13. Gentzen, G.: Untersuchungen über das logische Schließen. *Mathematische Zeitschrift* 39, 176–210 (1934)
14. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming. *Journal of Logic and Computation* 2(6), 719–770 (1992)
15. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming* 5, 235–324 (1998)
16. Lamport, L.: The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16(3), 872–923 (1994)

17. Lamport, L.: A summary of TLA+ (2000)
18. Liu, Y.A., Stoller, S.D.: Role-based access control: A corrected and simplified specification. In: Wang, C., King, S., Wachter, R., Herklotz, R., Arney, C., Toth, G., Hislop, D., Heise, S., Combs, T. (eds.) Department of Defense Sponsored Information Security Research: New Methods for Protecting Against Cyber Threats. Wiley (2007)
19. Mayer, M.C., Pirri, F.: First order abduction via tableau and sequent calculi. *Logic Journal of the Interest Group in Pure and Applied Logics* 1(1), 99–117 (1993)
20. Merz, S.: The specification language TLA+. *Logics of Specification Languages* pp. 401–451 (2008)
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer-Verlag (2002)
22. Paulson, L.C.: A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science* 5(3), 73–87 (1999)
23. Russo, A., Miller, R., Nusheibeh, B., Kramer, J.: An abductive approach for analysing event-based requirements specification. In: International Conference on Logic Programming. pp. 22–37. *Lecture Notes in Computer Science*, Springer-Verlag (2002)
24. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. *IEEE Computer* 29(2), 38–47 (1996)
25. Soler-Toscano, F., Nepomuceno-Fernández, A., Aliseda-Llera, A.: Abduction via C -tableaux and δ -resolution. *Journal of Applied Non-Classical Logics* 19(2), 211–225 (2009)