Policy automata for stateful authorization

Massimiliano Baldo[1,2], Marino Miculan[2] and Matteo Paier[1,2]

[1]IMT School for Advanced Studies Lucca, Piazza San Francesco 19, 55100 Lucca, Italy
[2]Department of Mathematics, Computer Science and Physics (DMIF), University of Udine, Via delle Scienze 206, 33100 Udine, Italy

Abstract

Modern policy languages decide whether an action is allowed by evaluating logical conditions over the request and the surrounding context, and by attaching a result (e.g. permit or forbid) to each matching rule. Yet most of these languages are *stateless*: each request is evaluated in isolation, which makes history-dependent (*usage*) policies awkward to express. In this paper we first introduce a minimal language of *policies with effects*, where a policy is a guarded effect: a guard is a logical formula on the current state and the effect is a generic state transformer fired when the guard holds. We then define *policy automata*, a deterministic, stateful variant of usage automata in which the policy state is explicit and transitions update it through effects. Each transition also *emits* an access-control decision, so a policy automaton reads as a Mealy machine whose output is the verdict. Determinism removes the need to instantiate the automaton over all resource assignments and makes run-time monitoring conceptually simple. We show the model is analysable over a decidable guard logic (well-formedness, reachability of a denial, and policy subsumption and equivalence are decidable) and that it is compositional: compliant languages are closed under conjunction and disjunction, with a "deny-overriding" rule-set union recovering the conjunction. The explicit state also pays off in expressiveness: policy automata are strictly more expressive than the stateless core of Cedar and Rego, with a memory cost that grows with the usage bound.

Keywords

automata theory, security and trust; specification and verification, policy languages, run-time monitoring.

1. Introduction

Controlling *how* resources are used, and not merely *whether* they may be accessed, is a recurring requirement in security: a file may be read only if it has been opened and not yet closed, a credential may be used at most a bounded number of times, two datasets in the same conflict-of-interest class may not both be accessed, and so on. Policies of this kind are *history dependent*: the verdict on an action depends on the sequence of actions performed so far.

Several automata models target this kind of stateful, data-aware specification. *Usage automata* [1, 2] extend finite state automata with edges carrying universally quantified resource variables and *guards*; their semantics is given by instantiating the variables to actual resources, yielding a family of finite state automata whose "offending" states denote violations. More expressive models keep an explicit memory: *register automata* [3] and *symbolic register automata* [4] process data words over an infinite alphabet by storing values in registers and testing them in the transition guards; *extended finite state machines* (EFSM) [5] equip a finite control with state variables, guarded transitions, and assignments; and, in runtime verification, *quantified event automata* [6] monitor parametric event traces.

Meanwhile, industrial *policy languages* have become widespread. Cedar [7] is the language behind AWS Verified Permissions, aimed at application-level authorization, while Rego is the language of the Open Policy Agent [8], a CNCF project used for cloud-native authorization such as Kubernetes admission control and microservice APIs. Both evaluate a request by means of logical rules and combining their result: in Cedar, every policy is tagged either `permit` or `forbid`, and `forbid` overrides `permit`. These languages are deliberately *stateless*: a decision is a pure function of the current request and context, which is excellent for analysability but makes genuinely history-dependent (usage) policies hard to express directly. Recent extensions to these languages, such as *OWSM* [9] for Rego/OPA and *Strobilus* [10] for Cedar, aim to address this gap by equipping policies with stateful effects.

ICTCS 2026: Italian Conference on Theoretical Computer Science, September 7–9, 2026, Udine, Italy
massimiliano.baldo@imtlucca.it (M. Baldo); marino.miculan@uniud.it (M. Miculan); matteo.paier@uniud.it (M. Paier)
0009-0001-9572-2692 (M. Baldo); 0000-0003-0755-3444 (M. Miculan); 0009-0000-7588-7169 (M. Paier)

This paper attacks the same gap from the theory side. We take the "guard + effect" structure of stateful policy languages and cast it as an automaton with explicit *state variables* and *symbolic, guarded transitions* (in the EFSM/symbolic register automata tradition) where the transition's *update* is the policy effect. We isolate a deterministic, effectful fragment of symbolic stateful automata tailored to policy enforcement, and study its *decidability*, *compositionality*, and *expressive power*.

Our contributions are:

- a *minimal policy language with effects* (Section 2), in which a policy is a set of guarded effects $\alpha(\bar{x}) : g \Rightarrow u$: the guard g is a logical formula over the current state and the effect u is a generic state transformer applied when the guard holds;
- *policy automata* (Section 3), an instance of the EFSM / symbolic register automata [5, 4] tailored to policy enforcement: registers, guarded transitions whose updates are the effects, and a per-transition decision (Mealy-style) emitting the access-control verdict. Well-formed policy automata, such as those induced by policies, are deterministic (Proposition 3.3) and recognise safety properties (Proposition 3.7), hence can be executed directly as monitors;
- *decidable analyses* (Section 4): over a decidable guard logic, well-formedness (Proposition 3.4) is decidable and for finitary automata so are reachability of a denial (Proposition 4.2) and policy subsumption and equivalence (Proposition 4.4)—determinism being exactly what keeps the symbolic case tractable (Remark 4.5);
- *closure properties* (Section 5): compliant languages are closed under conjunction and disjunction (Propositions 5.1 and 5.2), and the natural "rule-set union with deny-overrides" is exactly conjunction (Proposition 5.4), pinning down the meaning of Cedar-style rule-combining algorithms;
- an *expressiveness* result (Section 6): effects make policy automata strictly more expressive than their stateless, Cedar/Rego-core fragment (Proposition 6.2), with a memory requirement that provably grows with the usage bound (Proposition 6.3).

Keeping the effect generic lets the same framework subsume both the boolean permit/forbid decisions of access control and the counters, flags, and sets needed by usage control. Determinism, in turn, replaces the universal instantiation over resources of usage automata [1] with a single, explicit run, which is exactly what a run-time monitor performs.

2. A minimal policy language with effects

We fix a finite set of *actions* $\alpha, \beta, \cdots \in \mathsf{Act}$, each with an *arity* $|\alpha| \in \mathbb{N}$, and a (possibly infinite) set of *values* $v, v', \cdots \in \mathsf{Val}$ on which actions operate (the *resources* of usage automata). An *event* $\alpha(v_1, \ldots, v_k) \in \mathsf{Ev}$ is the firing of an action α with $k = |\alpha|$ on values $v_1, \ldots, v_k$. A *trace* $\eta = e_1 e_2 \cdots e_n \in \mathsf{Ev}^*$ is a finite sequence of events.

States and guards. Unlike usage automata, whose control states are abstract, our policies carry an explicit *state*. Let X be a finite set of *registers*; a *state* $s \in \Sigma$ is a map $s : \mathsf{X} \to \mathsf{Val}$ assigning a value to each register (we write $\Sigma = \mathsf{Val}^{\mathsf{X}}$). Registers record the relevant history, e.g. a counter of past reads or the set of datasets already accessed.

Guards and effects may refer both to the registers in X and to the formal parameters of the firing action, drawn from a set Var of *variables* (disjoint from X). A *binding* $\theta : \mathsf{Var} \rightharpoonup \mathsf{Val}$ instantiates the parameters with the values carried by an event; we write $\Theta = (\mathsf{Val} + 1)^{\mathsf{Var}}$ for the space of bindings.

Definition 2.1 (Guards)**.** The set G of *guards* is generated by a logic over terms $t ::= v \mid x \mid \mathsf{r} \mid \ldots$ (values $v \in \mathsf{Val}$, variables $x \in \mathsf{Var}$, registers $\mathsf{r} \in \mathsf{X}$, and any application-specific operations), as

$$g ::= \mathit{true} \;\mid\; t = t' \;\mid\; p(t_1, \ldots, t_m) \;\mid\; \neg g \;\mid\; g \wedge g,$$

where p ranges over a fixed set of interpreted predicates (equality, ordering, membership, …). We write $g \vee g'$, $t \neq t'$, and $g \Rightarrow g'$ as the usual abbreviations.

The logic is a parameter of the framework: instantiating it with quantifier-free equality recovers the guards of usage automata [1], while richer theories (arithmetic, sets) match the expressions found in Cedar and Rego conditions.

Given a state s and a binding θ, a closed term t denotes a value $[\![t]\!]_{s,\theta} \in \mathsf{Val}$ in the obvious way ($[\![v]\!] = v$, $[\![x]\!]_{s,\theta} = \theta(x)$, $[\![\mathsf{r}]\!]_{s,\theta} = s(\mathsf{r})$). *Satisfaction* $s, \theta \models g$ is then standard:

$$s, \theta \models \mathit{true}; \quad s, \theta \models t = t' \text{ iff } [\![t]\!]_{s,\theta} = [\![t']\!]_{s,\theta}; \quad s, \theta \models \neg g \text{ iff } s, \theta \not\models g;$$

and conjunction and interpreted predicates are defined as expected. When g has no variables we simply write $s \models g$.

Effects. The second component of a policy is the *effect*, which we keep generic: it is an operation that, fired by an event, transforms the state.

Definition 2.2 (Effects)**.** An *effect* is a symbol $u \in \mathsf{U}$ with an interpretation $[\![u]\!] : \Sigma \times \Theta \to \Sigma$, mapping the current state and the values carried by the firing event to the updated state.

We do not commit to a concrete syntax for effects. A typical instance lets an effect be a parallel *register assignment* $\langle \mathsf{r}_1 := t_1, \ldots, \mathsf{r}_n := t_n \rangle$, with $[\![\langle \mathsf{r}_i := t_i \rangle]\!](s, \theta) = s[\mathsf{r}_i \mapsto [\![t_i]\!]_{s,\theta}]$, while the identity effect skip leaves the state unchanged.

Decisions. The access-control *verdict* of an event is not a side effect on the state but an observable *output*, in the spirit of Cedar and Rego, where each request yields a decision. We therefore fix a set D of *decisions* with a designated *default* $d_0 \in \mathsf{D}$; the canonical instance is $\mathsf{D} = \{\mathtt{permit}, \mathtt{deny}\}$ with $d_0 = \mathtt{deny}$ (default-deny), but D may be richer (e.g. decisions carrying obligations, or explanations). A distinguished subset $\mathsf{D}_{\mathtt{deny}} \subseteq \mathsf{D}$ marks the *denials*, i.e. the decisions whose emission counts as a violation. We call *decision domain* the triple $(\mathsf{D}, \mathsf{D}_{\mathtt{deny}}, d_0)$.

Policies. With states, guards, effects, and decisions in hand, a *policy* can now be assembled. Conceptually, a policy is a finite collection of *rules*: each rule binds an action to a guarded reaction, prescribing the decision to emit and the effect to apply whenever the guard holds in the current state. Together with an initial state and a default decision for events that match no rule, this collection constitutes a policy.

Definition 2.3 (Rules and policies)**.** Let $(\mathsf{D}, \mathsf{D}_{\mathtt{deny}}, d_0)$ be a fixed decision domain. A *rule* has the form

$$\alpha(x_1, \ldots, x_k) : g \;\Rightarrow\; d \,;\, u,$$

where $\alpha \in \mathsf{Act}$ with $|\alpha| = k$, the $x_i \in \mathsf{Var}$ are distinct, $g \in \mathsf{G}$ is a guard whose variables are among $x_1, \ldots, x_k$, $d \in \mathsf{D}$ is the emitted *decision*, and $u \in \mathsf{U}$ is an effect. A *policy* is a triple $P = \langle R, s_0, d_0 \rangle$, where R is a finite set of rules, $s_0 \in \Sigma$ is the *initial state*, and $d_0 \in \mathsf{D}$ is the default decision.

Intuitively, when an event $\alpha(\bar{v})$ occurs, the rule for α whose guard holds in the current state *fires*: it *emits* its decision d and applies its effect u to obtain the next state. If no rule fires, the default decision d_0 is emitted and the state is left unchanged.

Definition 2.4 (Enabled rules and one-step semantics)**.** Let $P = \langle R, s_0, d_0 \rangle$. For a state s and an event $\alpha(\bar{v})$ with $\bar{v} = v_1 \ldots v_k$, let $\theta = \{x_i \mapsto v_i\}$. The set of *enabled* rules is

$$\mathit{En}(s, \alpha(\bar{v})) \triangleq \{\, (\alpha(\bar{x}) : g \Rightarrow d \,;\, u) \in R \mid s, \theta \models g \,\}.$$

When $|\mathit{En}(s, \alpha(\bar{v}))| \leq 1$, the one-step relation $s \xrightarrow{\alpha(\bar{v})\,/\,d} s'$ *emits* a decision $d \in \mathsf{D}$ and moves to s':

$$s \xrightarrow{\alpha(\bar{v})\,/\,d} s' \quad \text{where} \quad (d, s') = \begin{cases} (d, [\![u]\!](s, \theta)) & \text{if } \mathit{En}(s, \alpha(\bar{v})) = \{\alpha(\bar{x}) : g \Rightarrow d \,;\, u\}, \\ (d_0, s) & \text{if } \mathit{En}(s, \alpha(\bar{v})) = \emptyset. \end{cases}$$

The empty-set case plays the role of the implicit self-loops added by the *complete* operator of usage automata [1]: an event matched by no rule leaves the state untouched and yields the default decision, rather than blocking. We deliberately leave the relation *undefined* when two or more rules are enabled; Section 3 discusses how to recover a total function, either by well-formedness or by conflict resolution.

Example 2.5 (Bounded usage)**.** Consider a file that must be *opened* before being *read*, may be read at most N times per session, and is reset on *close*. Take $\mathsf{Act} = \{open, read, close\}$ (all of arity 0), registers $\mathsf{X} = \{\mathsf{open}, \mathsf{n}\}$ with $s_0 = \{\mathsf{open} \mapsto 0, \mathsf{n} \mapsto 0\}$, and $\mathsf{D} = \{\mathtt{permit}, \mathtt{deny}\}$ with default $d_0 = \mathtt{permit}$. The rules are

$$
\begin{aligned}
&open() : true \;\Rightarrow\; \mathtt{permit} \,;\, \langle \mathsf{open} := 1 \rangle \\
&read() : \mathsf{open} = 1 \wedge \mathsf{n} < N \;\Rightarrow\; \mathtt{permit} \,;\, \langle \mathsf{n} := \mathsf{n} + 1 \rangle \\
&read() : \mathsf{open} = 0 \vee \mathsf{n} \geq N \;\Rightarrow\; \mathtt{deny} \,;\, \mathsf{skip} \\
&close() : true \;\Rightarrow\; \mathtt{permit} \,;\, \langle \mathsf{open} := 0,\ \mathsf{n} := 0 \rangle
\end{aligned}
$$

The two *read* rules have mutually exclusive guards, so at most one is ever enabled. A read before any open, or an $(N{+}1)$-th read, fires the third rule and emits `deny` without touching the state; every other event is permitted. □

3. Policy automata

We now give the automaton model. Rather than the abstract control states of usage automata, a *policy automaton* has a finite control *plus* a vector of state variables, and its edges are symbolic transitions guarded by formulas on those variables, in the style of extended finite state machines [5] and symbolic register automata [4]. A configuration is a pair (q, s) of a control location q and a register valuation s.

Definition 3.1 (Policy automaton)**.** Let $(\mathsf{D}, \mathsf{D}_{\mathtt{deny}}, d_0)$ be a fixed decision domain. A *policy automaton* is a tuple $\mathcal{A} = \langle Q, q_0, \mathsf{X}, s_0, T \rangle$, where Q is a finite set of *control locations*, $q_0 \in Q$ is the initial location, X is a finite set of registers with state space $\Sigma = \mathsf{Val}^{\mathsf{X}}$ and initial valuation s_0, and T is a finite set of *transitions*

$$q \xrightarrow{\alpha(\bar{x}) \,:\, g \,\Rightarrow\, d;u} q',$$

with $q, q' \in Q$, $\alpha \in \mathsf{Act}$, $\bar{x}$ the formal parameters of α, $g \in \mathsf{G}$, emitted decision $d \in \mathsf{D}$, and effect $u \in \mathsf{U}$.

A transition $q \xrightarrow{\alpha(\bar{x}) \,:\, g \,\Rightarrow\, d;u} q'$ is *enabled* in configuration (q, s) on event $\alpha(\bar{v})$, with $\theta = \{\bar{x} \mapsto \bar{v}\}$, when $s, \theta \models g$. The *configuration step* then emits a decision $d \in \mathsf{D}$,

$$(q, s) \xRightarrow{\alpha(\bar{v}) \,/\, d} (q'', s''), \quad \text{where} \quad (q'', s'', d) = \begin{cases} (q', [\![u]\!](s, \theta), d) & \text{if } q \xrightarrow{\alpha(\bar{x}) : g \Rightarrow d;u} q' \text{ is enabled,} \\ (q, s, d_0) & \text{if no transition is enabled.} \end{cases}$$

The automaton is *deterministic* if in every configuration each event enables at most one transition.

The reading is literal: "on input $\alpha(\bar{v})$, if guard g holds of the current registers, emit the decision d, move to q', and update the registers by u". When no transition fires the default decision d_0 is emitted and the configuration is unchanged, recovering the implicit self-loops of usage automata. An event is *denied* exactly when the decision it emits lies in $\mathsf{D}_{\mathtt{deny}}$. Note that transitions in T are syntactic objects relating two locations, while the semantics acts on configurations, $(q, s) \Rightarrow (q', s')$: a trace is still a sequence of events, and it is the *run* it induces (Definition 3.6) that is a sequence of configurations.

From policies to automata. A policy of Section 2 is the single-location case of this model: its rules become self-transitions $q_0 \xrightarrow{\alpha(\bar{x}) : g \Rightarrow d;u} q_0$ on the only location q_0. Explicit locations are convenient when a policy goes through distinct *phases*, but are not essential: a location can always be encoded as a finite-domain register, so the embedding is essentially onto (Proposition 3.5). Conversely, determinism of the induced automaton is exactly the well-formedness of the policy.

Definition 3.2 (Well-formedness)**.** A policy P (resp. a policy automaton) is *well-formed* if for every location q, action α, valuation s, and binding θ, at most one transition is enabled; equivalently, the guards of any two transitions leaving q on α are mutually unsatisfiable.

Proposition 3.3 (Determinism)**.** *If P is well-formed, the configuration step is a total function on $\mathcal{C} = Q \times \Sigma$, so the induced automaton $\mathcal{A}_P$ is deterministic.*

Proof sketch. By Definition 3.2 at most one transition is enabled in each (q, s) on each event and the step is otherwise the identity; hence it is defined and single-valued everywhere, i.e. a total function $\mathcal{C} \times \mathsf{Ev} \to \mathcal{C}$. □

Well-formedness reduces to satisfiability in the guard logic, which Cedar and Rego already keep decidable for analysis.

Proposition 3.4 (Deciding well-formedness)**.** *If the guard logic has decidable satisfiability, then it is decidable whether a policy automaton $\mathcal{A}$ is well-formed, using at most one satisfiability query per pair of transitions sharing a source location and action.*

Proof. By Definition 3.2, $\mathcal{A}$ is *not* well-formed exactly when some location q and action α carry two distinct transitions $q \xrightarrow{\alpha(\bar{x}):g \Rightarrow d\,;u} q'$ and $q \xrightarrow{\alpha(\bar{x}):g' \Rightarrow d'\,;u'} q''$ (sharing the formal parameters $\bar{x}$, up to renaming) whose guards are jointly satisfiable, i.e. $s, \theta \models g \wedge g'$ for some state s and binding θ. The guard logic is closed under conjunction, so $g \wedge g'$ is itself a guard and its satisfiability is decided by the assumed oracle. There are at most $\binom{|T|}{2}$ such pairs, so testing them all decides well-formedness. □

The embedding of policies into automata also has a converse: every automaton is, up to a location register, a flat policy. Synthesising the rules that induce a given automaton is thus immediate and the two presentations are interchangeable.

Proposition 3.5 (Flattening)**.** *Suppose the effect language is closed under adding a register assignment. Then for every policy automaton $\mathcal{A} = \langle Q, q_0, \mathsf{X}, s_0, T \rangle$ there is a single-location policy $P_{\mathcal{A}}$ over the registers $\mathsf{X} \uplus \{\mathsf{loc}\}$, with loc ranging over Q, such that $\mathcal{A}_{P_{\mathcal{A}}}$ and $\mathcal{A}$ emit the same decision on every trace; moreover $P_{\mathcal{A}}$ is well-formed iff $\mathcal{A}$ is. So policies and policy automata define the same decision functions.*

Proof. Take loc with initial value q_0 and turn each transition $q \xrightarrow{\alpha(\bar{x}):g \Rightarrow d\,;u} q'$ into the rule $\alpha(\bar{x}) : (\mathsf{loc} = q \wedge g) \Rightarrow d\,;(u,\ \mathsf{loc} := q')$, with initial state $s_0[\mathsf{loc} \mapsto q_0]$ and default d_0. The bijection $(q, s) \leftrightarrow (s[\mathsf{loc} \mapsto q])$ between configurations of $\mathcal{A}$ and of $\mathcal{A}_{P_{\mathcal{A}}}$ commutes with the configuration step and preserves the emitted decision: the guard $\mathsf{loc} = q$ selects exactly the rules of location q and the assignment $\mathsf{loc} := q'$ mirrors the change of location. So the two emit the same decisions on every trace. For well-formedness, two rules for α are jointly enabled iff $(\mathsf{loc} = q \wedge g) \wedge (\mathsf{loc} = q' \wedge g')$ is satisfiable, which forces $q = q'$ and reduces to the joint satisfiability of g, g' within a single location of $\mathcal{A}$. □

When a policy is not well-formed, determinism is recovered by a *conflict-resolution* operator that orders the enabled transitions and combines their effects: Cedar's "`forbid` overrides `permit`" is one such operator over the boolean decision.

Definition 3.6 (Run and compliance)**.** Let $\mathcal{A}$ be a deterministic policy automaton and $\eta = e_1 \cdots e_n$ a trace. The *run* of η is the unique sequence $(q_0, s_0) \xrightarrow{e_1/d_1} (q_1, s_1) \cdots \xrightarrow{e_n/d_n} (q_n, s_n)$ of configuration steps, producing the decisions $d_1 \cdots d_n \in \mathsf{D}^*$. The trace η *complies* with $\mathcal{A}$, written $\eta \models \mathcal{A}$, iff $d_i \notin \mathsf{D}_{\mathsf{deny}}$ for all $i \in 1..n$; otherwise it *violates* $\mathcal{A}$, the first denied event being the offending one.

Since the step is a function, each trace has exactly one run, so compliance is decided by a single deterministic pass. This is in contrast with usage automata [1], where a trace is checked against the family $\{\mathcal{A}_\varphi(\sigma, \mathsf{Val})\}_\sigma$ of all resource instantiations.

Proposition 3.7 (Safety)**.** *The* compliant language $L(\mathcal{A}) \triangleq \{\eta \mid \eta \models \mathcal{A}\}$ *is prefix closed: if $\eta\eta' \models \mathcal{A}$ then $\eta \models \mathcal{A}$. Hence policy automata recognise* safety *properties.*

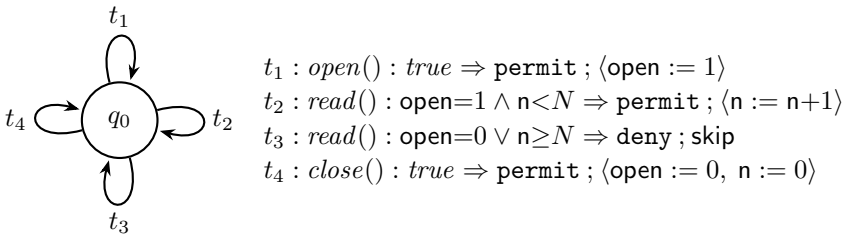

Figure 1: The single-location policy automaton induced by the bounded-usage policy of Example 2.5, with registers $\{\mathsf{open}, \mathsf{n}\}$ and initial valuation $\mathsf{open} = 0,\ \mathsf{n} = 0$. The four self-loops $t_1, \ldots, t_4$ correspond, in order, to the four rules of the policy.

Proof. The run of η is a prefix of the run of $\eta\eta'$, because the step is deterministic; if no decision emitted along the longer run is a denial, none emitted along the prefix is either. □

The converse, that is, demanding that something *eventually* happen, falls outside the model, by the same argument.

Example 3.8 (Liveness is out of reach)**.** With the actions $\mathsf{Act} = \{open, read, close\}$ of Example 2.5, consider the liveness property "the file is eventually read at least once",

$$L_{\text{live}} \triangleq \{\, \eta \in \mathsf{Ev}^* \mid \text{some event of } \eta \text{ is } read \,\}.$$

No policy automaton recognises it. The empty trace ε is a prefix of the trace $read$ which belongs to L_{live}, so by Proposition 3.7 a policy automaton with $L(\mathcal{A}) = L_{\text{live}}$ would have $\varepsilon \in L_{\text{live}}$, yet ε contains no $read$ event. The same argument rules out every property $\Pi \subseteq \mathsf{Ev}^*$ with $\varepsilon \notin \Pi \neq \emptyset$, and more generally any property that is not prefix closed. □

Policy automata are thus, by design, a safety-only framework: they can forbid what should not happen but cannot demand that something eventually does. Enforcing such positive obligations would require additional machinery (e.g. timeouts or obligation effects) outside the scope of this paper.

Remark 3.9 (Monitoring). A deterministic policy automaton is directly executable as a run-time monitor: it keeps the current configuration (q, s) and, on each event e, updates it by the configuration step and outputs the emitted decision d, denying the event when $d \in \mathsf{D}_{\texttt{deny}}$. The space required is that of one configuration, i.e. of the finite control plus the registers X; when the registers range over a bounded domain the monitor uses bounded memory. This is the explicit-state counterpart of the bounded-space enforcement mechanism of usage automata, obtained here for free from determinism.

Example 3.10 (The bounded-usage automaton)**.** The policy of Example 2.5 is well-formed: the $read$ guards $\mathsf{open} = 1$ and $\mathsf{open} = 0$ are mutually unsatisfiable. Hence, it induces a single-location, deterministic policy automaton over registers $\{\mathsf{open}, \mathsf{n}\}$ (Fig. 1). When run as a monitor, it emits `permit` throughout $open\ read^N$ and denies $open\ read^{N+1}$, whose last step fires transition t_3 and emits `deny`, using only the counter n and the flag open. □

4. Deciding policy analysis

Recall $L(\mathcal{A}) = \{\eta \mid \eta \models \mathcal{A}\}$, the prefix-closed set of traces $\mathcal{A}$ deems compliant (Proposition 3.7). Two questions recur in practice: does a policy ever deny, i.e. is a denying rule reachable at all, and is one policy at least as permissive as another? Both are reachability problems over the configuration space, decidable if that space is effectively finite. Call $\mathcal{A}$ *finitary* if the set $\mathrm{Reach}(\mathcal{A}) \subseteq \mathcal{C}$ of configurations occurring in the run of some trace is finite and computable; this holds, for instance, when the registers range over a fixed finite domain that the effects preserve. (Determinism is not assumed here.)

Lemma 4.1 (Local denial test)**.** *If the guard logic has decidable satisfiability, then for any configuration (q, s) it is decidable whether some event $\alpha(\bar{v})$ makes the step from (q, s) emit a denial $d \in \mathsf{D}_{\texttt{deny}}$.*

Proof. There are finitely many actions $\alpha \in \mathsf{Act}$. For each (q, s), it denies some $\alpha(\bar{v})$ in one of two ways. Either an enabled transition denies: for a transition $q \xrightarrow{\alpha(\bar{x}):g \Rightarrow d;u} q'$ with $d \in \mathsf{D}_{\mathtt{deny}}$, this happens iff g is satisfiable at s over the parameters, i.e. $\exists\theta.\, s, \theta \models g$. Or no transition is enabled and the default denies: when $d_0 \in \mathsf{D}_{\mathtt{deny}}$, this happens iff $\bigwedge_t \neg g_t$ is satisfiable at s, where t ranges over the transitions leaving q on α. Each is a satisfiability query with s fixed, decided by the assumed oracle; the test is their finite disjunction. □

Proposition 4.2 (Reachability of a denial)**.** *For a finitary policy automaton over a guard logic with decidable satisfiability, it is decidable whether $\mathcal{A}$ admits a violating trace (i.e., whether $L(\mathcal{A}) \neq \mathsf{Ev}^*$).*

Proof. Compute $\mathrm{Reach}(\mathcal{A})$ by forward fixpoint from (q_0, s_0): the configuration step is computable (guards are decidable, effects are functions) and $\mathrm{Reach}(\mathcal{A})$ is finite, so the iteration terminates. By Lemma 4.1, test each reached configuration for a denying event. A violating trace exists iff some reachable configuration passes the test: the trace reaching it, extended by the denying event, violates $\mathcal{A}$; conversely every violating trace exhibits such a configuration. □

A denying rule that no reachable configuration can ever fire is dead code and the same procedure detects it. Comparing two policies is the product of this construction.

Definition 4.3 (Subsumption)**.** For policy automata $\mathcal{A}_1, \mathcal{A}_2$ over the same actions, $\mathcal{A}_1$ is *subsumed by* $\mathcal{A}_2$, written $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$, if $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$—every trace $\mathcal{A}_1$ accepts, $\mathcal{A}_2$ accepts too, so $\mathcal{A}_1$ is at least as restrictive. They are *equivalent* if $L(\mathcal{A}_1) = L(\mathcal{A}_2)$.

Proposition 4.4 (Deciding subsumption and equivalence)**.** *For finitary $\mathcal{A}_1, \mathcal{A}_2$ over a guard logic with decidable satisfiability, both $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ and $\mathcal{A}_1 \equiv \mathcal{A}_2$ are decidable.*

Proof. Form the product policy automaton with locations $Q_1 \times Q_2$ and registers $\mathsf{X}_1 \uplus \mathsf{X}_2$, so its configurations are $((q_1, q_2), (s_1, s_2))$ and a step on event $\alpha(\bar{v})$ runs both components in parallel; its reachable set is contained in $\mathrm{Reach}(\mathcal{A}_1) \times \mathrm{Reach}(\mathcal{A}_2)$, hence finite and computable. Subsumption fails iff some trace lies in $L(\mathcal{A}_1) \setminus L(\mathcal{A}_2)$. Take such an η and the first event at which $\mathcal{A}_2$ denies: along the prefix before it both components only permit (so the prefix is in $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$), while on that event $\mathcal{A}_1$ permits ($\eta \in L(\mathcal{A}_1)$) and $\mathcal{A}_2$ denies. Conversely any such configuration yields a trace in $L(\mathcal{A}_1) \setminus L(\mathcal{A}_2)$. So restrict the product reachability to configurations reached through events on which *both* components permit (Lemma 4.1 componentwise), and at each such configuration test (again a satisfiability query, with both register valuations fixed) for an event on which $\mathcal{A}_1$ permits and $\mathcal{A}_2$ denies. Finiteness makes the search terminate, deciding $\sqsubseteq$; equivalence is $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2 \wedge \mathcal{A}_2 \sqsubseteq \mathcal{A}_1$. □

Remark 4.5 (Beyond the finitary case, and the role of determinism). When registers range over an infinite domain, reachability of a denial is emptiness, and subsumption is language inclusion, of the symbolic register automata underlying the policies. Emptiness is decidable for the register theories where these automata enjoy it [4], but undecidable in general—two unbounded counters already encode Minsky machines. Inclusion is more delicate: it is decidable for the *deterministic* fragment we consider [4] but undecidable for nondeterministic register automata [3]. Well-formedness (Proposition 3.4) is exactly what keeps policies in that decidable, deterministic fragment.

5. Combining policies

Policies are rarely written in isolation: groups of administrators, layered organisational rules, and multiple stakeholders all give rise to combinations. Two such combinations are natural: a request must satisfy *both* policies, or it must satisfy *at least one*. Both are realised within the model.

Proposition 5.1 (Closure under conjunction)**.** *For well-formed finitary $\mathcal{A}_1, \mathcal{A}_2$ over the same actions and decision domain, there is a well-formed finitary policy automaton $\mathcal{A}_1 \times \mathcal{A}_2$, the* product*, with $L(\mathcal{A}_1 \times \mathcal{A}_2) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.*

Proof. Take locations $Q_1 \times Q_2$, registers $\mathsf{X}_1 \uplus \mathsf{X}_2$ and initial configuration $((q_0^1, q_0^2), (s_0^1, s_0^2))$. By determinism (Proposition 3.3) each component has a unique configuration step $(q_i, s_i) \xrightarrow{\alpha(\bar{v})/d_i} (q_i', s_i')$ on event $\alpha(\bar{v})$; the product step is $((q_1, q_2), (s_1, s_2)) \xrightarrow{\alpha(\bar{v})/d_\times} ((q_1', q_2'), (s_1', s_2'))$, with the deny-overriding combination $d_\times \in \mathsf{D}_{\mathtt{deny}}$ iff $d_1 \in \mathsf{D}_{\mathtt{deny}}$ or $d_2 \in \mathsf{D}_{\mathtt{deny}}$. Disjointness of $\mathsf{X}_1, \mathsf{X}_2$ makes the component effects commute, so the step is realised by transitions whose guards conjoin the component guards, whose decision is $d_\times$ and whose effect is the parallel composition (with each component's default fall-through handled by its implicit d_0-emitting self-loop). Two such product transitions jointly enabled at a configuration would force two component transitions jointly enabled in $\mathcal{A}_1$ or in $\mathcal{A}_2$, contradicting their well-formedness; the product is therefore well-formed. Reachability is contained in $\mathrm{Reach}(\mathcal{A}_1) \times \mathrm{Reach}(\mathcal{A}_2)$, hence finite. Finally, $\eta \in L(\mathcal{A}_1 \times \mathcal{A}_2)$ iff $d_{\times,i} \notin \mathsf{D}_{\mathtt{deny}}$ for all i, iff $d_{1,i} \notin \mathsf{D}_{\mathtt{deny}}$ and $d_{2,i} \notin \mathsf{D}_{\mathtt{deny}}$ for all i, iff $\eta \in L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$. □

This is exactly the construction used informally in the proof of Proposition 4.4; we extract it here as a closure result.

Proposition 5.2 (Closure under disjunction)**.** *For well-formed finitary $\mathcal{A}_1, \mathcal{A}_2$ as above, there is a well-formed finitary policy automaton $\mathcal{A}_1 + \mathcal{A}_2$ with $L(\mathcal{A}_1 + \mathcal{A}_2) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$.*

Proof. Augment the product $\mathcal{A}_1 \times \mathcal{A}_2$ with two boolean registers f_1, f_2 initialised to 0, and overlay the following monitor on each step: define $f_i' = 1$ if $f_i = 1$ or the $\mathcal{A}_i$-component would emit a denial, else $f_i' = 0$; emit `deny` iff $f_1' = f_2' = 1$, and `permit` otherwise; commit $f_i := f_i'$. Splitting each product transition according to the four values of (f_1, f_2) realises this within the model; reachability grows by at most a factor of 4, so finitariness is preserved, as is well-formedness (the split is deterministic). Let τ_i be the first index at which the run of η in $\mathcal{A}_i$ would emit a denial (∞ otherwise); then $f_i = 1$ after step k iff $k \geq \tau_i$, so the merged step at k emits `deny` iff $k \geq \max(\tau_1, \tau_2)$. The merged run of η avoids denial iff $\max(\tau_1, \tau_2) > |\eta|$, iff η avoids denial in $\mathcal{A}_1$ or in $\mathcal{A}_2$, iff $\eta \in L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$. □

The two constructions also answer, at the policy level, what happens when one *merges rule sets*—the most direct way administrators combine policies in practice. Combining rule sets forces a choice of how to fuse the decisions the two sides emit. Call a binary operation $\oplus : \mathsf{D} \times \mathsf{D} \to \mathsf{D}$ a *deny-overriding combinator* when it is *compliance-respecting*, i.e.

$$d_1 \oplus d_2 \in \mathsf{D}_{\mathtt{deny}} \quad \text{iff} \quad d_1 \in \mathsf{D}_{\mathtt{deny}} \text{ or } d_2 \in \mathsf{D}_{\mathtt{deny}}.$$

Membership in $\mathsf{D}_{\mathtt{deny}}$ is all that compliance tests, so this is the only constraint the compliant language sees; the specific denial (or specific permit) returned is immaterial and left to the domain. In the canonical binary domain $\mathsf{D} = \{\mathtt{permit}, \mathtt{deny}\}$ the condition forces $\oplus$ uniquely (`deny` absorbing, `permit` neutral); with several denials or several non-denials any compliance-respecting choice will do (e.g. a fixed priority order on D resolving ties within $\mathsf{D}_{\mathtt{deny}}$ and within $\mathsf{D} \setminus \mathsf{D}_{\mathtt{deny}}$).

Merging is not the bare set union of rules, though: an event handled by a rule on one side but by *no* rule on the other is still decided on that other side—by its default d_0—and dropping this default is unsound. Under default-deny, for instance, a `permit` rule of P_1 facing a silent P_2 must yield a denial, since standalone P_2 denies the event and the trace already violates P_2. The construction therefore *completes each side with its default* before combining. For an action α and rule set R, let $\mathrm{en}_R^\alpha \triangleq \bigvee \{\, g \mid (\alpha(\bar{x}) : g \Rightarrow d\,;u) \in R \,\}$ be the guard holding exactly where some R-rule on α is enabled (at most one, by well-formedness).

Definition 5.3 (Deny-overriding rule-set union)**.** Let P_1, P_2 be well-formed policies sharing actions and a decision domain, with defaults $d_{0,1}, d_{0,2}$, and fix a deny-overriding combinator $\oplus$. Assume a parallel composition $u_1 \parallel u_2$ of effects is defined for every co-enabled cross-side rule pair (automatic for disjoint registers; characterised for shared ones in Definition 5.6). The *deny-overriding rule-set union* $P_1 \cup_{\mathtt{deny}} P_2$ is the policy with default $d_{0,1} \oplus d_{0,2}$ whose rules are, for every action α:

- for each cross-side pair co-enabled on α ($\alpha(\bar{x}) : g_1 \Rightarrow d_1 \,;\, u_1$ in R_1 and $\alpha(\bar{x}) : g_2 \Rightarrow d_2 \,;\, u_2$ in R_2) the joint rule
$$\alpha(\bar{x}) : g_1 \wedge g_2 \;\Rightarrow\; (d_1 \oplus d_2) \,;\, (u_1 \parallel u_2);$$
- for each rule $\alpha(\bar{x}) : g_1 \Rightarrow d_1 ; u_1$ of R_1, the one-sided residual $\alpha(\bar{x}) : g_1 \wedge \neg\, \mathrm{en}^{\alpha}_{R_2} \Rightarrow (d_1 \oplus d_{0,2}) ; u_1$, and symmetrically for each rule of R_2 (combined with $d_{0,1}$).

The joint rule and the two residuals partition, per action, the region where either side fires, so the merged policy stays well-formed; the construction collapses to the bare union $R_1 \cup R_2$ exactly when both defaults are non-denials, making $\oplus$ with $d_{0,j}$ neutral.

Proposition 5.4 (Rule-set union with deny-overrides is conjunction)**.** *Let P_1, P_2 be well-formed policies over disjoint register sets, sharing actions and decision domain. For any deny-overriding combinator $\oplus$, their deny-overriding rule-set union $P_1 \cup_{\mathtt{deny}} P_2$ is a well-formed policy whose induced automaton is, up to a location register, the product $\mathcal{A}_{P_1} \times \mathcal{A}_{P_2}$. Hence its compliant language is $L(P_1) \cap L(P_2)$.*

Proof. Disjointness of registers makes the parallel composition $u_1 \parallel u_2$ of any two co-enabled cross-side effects total, and co-enabled rules *within* R_1 or within R_2 are excluded by well-formedness. On each event the merged policy emits $\hat{d}_1 \oplus \hat{d}_2$, where $\hat{d}_j$ is the decision of the enabled R_j-rule if one fires and the default $d_{0,j}$ otherwise (this is precisely what the joint rules, the residuals, and the combined default $d_{0,1} \oplus d_{0,2}$ encode) and applies the corresponding effects in parallel. That is the configuration step of the product $\mathcal{A}_{P_1} \times \mathcal{A}_{P_2}$, whose deny-overriding combination is $\oplus$ (Proposition 5.1); the language identity follows. □

Remark 5.5 (Why the default completion is needed). Take default-deny ($d_0 = \mathtt{deny}$), let P_1 carry a single `permit` rule for an action and let P_2 have no rule on it. Standalone, P_2 denies the event, so it lies outside $L(P_2)$, hence outside $L(P_1) \cap L(P_2)$. The bare rule union $R_1 \cup R_2$ would fire only P_1's rule and *permit*, missing the intersection; the residual clause of Definition 5.3 instead emits $\mathtt{permit} \oplus d_{0,2} = \mathtt{permit} \oplus \mathtt{deny} = \mathtt{deny}$, restoring the identity.

The disjointness assumption is essential: with shared registers, two co-enabled cross-side effects can disagree on a common write, and one side's writes can flow into the other side's guards, derailing the standalone runs from the merged one. Both failure modes admit a static characterisation, which recovers the identity over shared registers.

We read off two pieces of standard data from the chosen guard and effect languages: for a guard g, the set $\mathrm{Read}(g) \subseteq \mathsf{X}$ of registers it mentions; for an effect u, the *write set* $\mathrm{Write}(u) = \{\mathsf{r} \in \mathsf{X} \mid \exists s, \theta.\ [\![u]\!](s,\theta)(\mathsf{r}) \neq s(\mathsf{r})\}$. For the parallel-assignment instance, $\mathrm{Write}(\langle \mathsf{r}_1 := t_1, \ldots, \mathsf{r}_k := t_k \rangle) = \{\mathsf{r}_1, \ldots, \mathsf{r}_k\}$ and $\mathrm{Write}(\mathsf{skip}) = \emptyset$. We extend Read to a rule (the registers occurring in its guard or in the right-hand sides of its effect) and to a rule set by union; similarly for Write.

Definition 5.6 (Compatibility)**.** Two well-formed policies P_1, P_2 over a shared register set X and decision domain are *compatible* when

1. (*non-interference*) for $i \neq j$, $\mathrm{Read}(R_i) \cap \mathrm{Write}(R_j) = \emptyset$;
2. (*effect-compatibility*) every pair of cross-side rules $\alpha(\bar{x}) : g_1 \Rightarrow d_1 \,;\, u_1$ in R_1 and $\alpha(\bar{x}) : g_2 \Rightarrow d_2 \,;\, u_2$ in R_2 on the same action agrees on every shared write, i.e. $s, \theta \models g_1 \wedge g_2$ implies $[\![u_1]\!](s,\theta)(\mathsf{r}) = [\![u_2]\!](s,\theta)(\mathsf{r})$ for all $\mathsf{r} \in \mathrm{Write}(u_1) \cap \mathrm{Write}(u_2)$.

Disjoint register sets imply both clauses trivially.

Under compatibility the parallel composition $u_1 \parallel u_2$ of any pair of co-enabled cross-side effects is well-defined, so Definition 5.3 applies verbatim to shared registers, and the disjoint-register identity of Proposition 5.4 extends.

Proposition 5.7 (Rule-set union under compatibility)**.** *Let P_1, P_2 be compatible well-formed policies sharing a register set and decision domain. For any deny-overriding combinator $\oplus$, their deny-overriding rule-set union $P_1 \cup_{\mathtt{deny}} P_2$ (Definition 5.3) is a well-formed policy with compliant language $L(P_1) \cap L(P_2)$.*

Proof. By effect-compatibility the parallel composition of any two co-enabled cross-side effects is well-defined, so the merged step is a deterministic function of configuration and event; intra-side co-enabling is ruled out by each P_i's well-formedness and cross-side co-enabling is resolved by the operator, so the union is well-formed.

For the language identity, run a trace η through P_1, P_2, and the union in parallel, and write $s_k^{(j)}, s_k^*$ for the standalone and merged states after k steps. We show by induction on k that s_k^* and $s_k^{(j)}$ agree on $\mathrm{Read}(R_j)$ for $j = 1, 2$. The base case is the shared initial state. For the step, registers in $\mathrm{Read}(R_j)$ are not written by $R_{j'}$ (non-interference), so along both runs they are modified solely by R_j's effects. By the inductive hypothesis P_j's guard evaluates the same way in the union as in the standalone at step $k+1$, so the same R_j-rule fires (or, if none, the residual/default clause of Definition 5.3 supplies $d_{0,j}$, matching P_j's standalone default), and its effect—which depends only on $\mathrm{Read}(R_j)$, again by non-interference—produces the same value on $\mathrm{Read}(R_j)$. Hence s_{k+1}^* and $s_{k+1}^{(j)}$ still agree there. Thus P_j's decision $\hat{d}_j$ at every step of the union matches its standalone decision, and the deny-overriding combination $\hat{d}_1 \oplus \hat{d}_2$ emits a denial iff some standalone run does: $\eta \in L(P_1 \cup_{\mathtt{deny}} P_2)$ iff $\eta \in L(P_1) \cap L(P_2)$. □

Notice that Proposition 5.4 is the special case of disjoint register sets.

Remark 5.8 (Deciding compatibility). Both clauses of Definition 5.6 are decidable when guards and effects are. Clause (i) is syntactic: $\mathrm{Read}(R_i)$ and $\mathrm{Write}(R_j)$ are finite, computable sets. Clause (ii), for each pair of cross-side rules sharing an action and each register $\mathsf{r} \in \mathrm{Write}(u_1) \cap \mathrm{Write}(u_2)$, is the unsatisfiability of $g_1 \wedge g_2 \wedge \big([\![u_1]\!](\cdot,\theta)(\mathsf{r}) \neq [\![u_2]\!](\cdot,\theta)(\mathsf{r})\big)$ in the combined theory of guards and effects—for parallel assignments $\mathsf{r} := t_1$ versus $\mathsf{r} := t_2$ this collapses to $g_1 \wedge g_2 \wedge t_1 \neq t_2$. Compatibility thus joins well-formedness (Proposition 3.4) as a static, SMT-discharged check on policies.

Example 5.9. Take $\mathsf{X} = \{\mathsf{n}\}$ shared, $\mathsf{Act} = \{\mathit{read}\}$ of arity 0, $s_0(\mathsf{n}) = 0$ and $d_0 = \mathtt{deny}$, with

$$P_1 = \{\, \mathit{read}() : \mathsf{n} < 3 \;\Rightarrow\; \mathtt{permit};\langle \mathsf{n} := \mathsf{n}{+}1 \rangle \,\}, \quad P_2 = \{\, \mathit{read}() : \mathsf{n} < 6 \;\Rightarrow\; \mathtt{permit};\langle \mathsf{n} := \mathsf{n}{+}2 \rangle \,\}.$$

Both are well-formed (single rule), $L(P_1) = L(P_2) = \mathit{read}^{\leq 3}$, so $L(P_1) \cap L(P_2) = \mathit{read}^{\leq 3}$. They violate effect-compatibility at $\mathsf{n} = 0$: both guards hold, yet u_1 writes $\mathsf{n} = 1$ and u_2 writes $\mathsf{n} = 2$. Resolving the effect conflict by always preferring u_1 gives a union with language $\mathit{read}^{\leq 3}$, but resolving by preferring u_2 gives $\mathit{read}^{\leq 2}$, strictly stricter than the intersection. Without clause (ii) of Definition 5.6 the deny-overriding union has no canonical, intersection-realising meaning. □

Remark 5.10 (The algebra of policies). Modulo equivalence (Definition 4.3), compliant languages of policy automata form a lattice under subsumption $\sqsubseteq$, with meet $\cap$ realised by the product (Proposition 5.1) and join $\cup$ realised by the deny-tracking product (Proposition 5.2). The top is Ev^* (permit everything), the bottom the all-deny policy. Complement is *not* in this lattice: by Proposition 3.7 compliant languages are safety properties and the complement of a safety property is in general a *co-safety* property, recognised by a different class of monitors. This sharpens the informal rule-combining tradition of Cedar [7] and similar access-control languages: their combining operators have a precise denotational meaning as Boolean operations on the compliant-trace languages, with deny-overrides matching the meet.

6. The role of state

The registers and effects are what separate policy automata from the history-insensitive core of Cedar and Rego. We make this precise. Call a policy automaton *stateless* if it has a single location and every effect is skip; then the configuration is (q_0, s_0) throughout and this is exactly the fragment in which a decision can depend only on the current event, not on the past.

Lemma 6.1. *For a stateless policy automaton $\mathcal{A}$ there is a function $\delta : \mathsf{Ev} \to \mathsf{D}$ such that, in the run of any trace, the decision emitted on an event e is $\delta(e)$, independent of its position and of the preceding events.*

Proof. Every configuration reached by $\mathcal{A}$ is (q_0, s_0), since the only location is q_0 and every effect leaves the state at s_0. The configuration step thus evaluates the guards always at s_0, so the emitted decision is a function of the event e alone; take it as $\delta(e)$. □

Proposition 6.2 (Effects add expressiveness)**.** *For every $N \geq 1$, the bounded-usage property of Example 2.5 is not $L(\mathcal{A})$ for any stateless policy automaton $\mathcal{A}$. Hence policy automata are strictly more expressive than their stateless, Cedar/Rego-core fragment.*

Proof. Suppose a stateless $\mathcal{A}$ had $L(\mathcal{A})$ equal to the bounded-usage property. The trace $\mathit{open}\ \mathit{read}$ is compliant (one read after an open is allowed, as $N \geq 1$), so every event along it is permitted by $\mathcal{A}$; by Lemma 6.1 this fixes $\delta(\mathit{open}) \notin \mathsf{D}_{\mathtt{deny}}$ and $\delta(\mathit{read}) \notin \mathsf{D}_{\mathtt{deny}}$. But then in $\mathit{open}\ \mathit{read}^{\,N+1}$ every event is permitted by $\mathcal{A}$ as well, so $\mathit{open}\ \mathit{read}^{\,N+1} \in L(\mathcal{A})$—whereas bounded usage denies its $(N{+}1)$-th read. Contradiction. The witness of Example 2.5 is a (stateful) policy automaton with this language, giving the strict inclusion. □

The gap is not merely about one location versus many: it is quantitative and forces memory that grows with N.

Proposition 6.3 (The counter is essential)**.** *Any deterministic policy automaton whose language is the bounded-usage property for parameter N has at least $N+1$ reachable configurations; if it is single-location, its registers take at least $N+1$ distinct valuations. No fixed finite register domain suffices for all N.*

Proof. Consider the prefixes $\eta_i = \mathit{open}\ \mathit{read}^{\,i}$ for $i = 0, \ldots, N$, each compliant and hence reaching a configuration c_i in the run. For $i < j$ the suffix $w = \mathit{read}^{\,N-i}$ distinguishes them: $\eta_i\, w = \mathit{open}\ \mathit{read}^{\,N}$ is compliant, while $\eta_j\, w = \mathit{open}\ \mathit{read}^{\,N+(j-i)}$ is not. Were $c_i = c_j$, determinism would make the runs of $\eta_i\, w$ and $\eta_j\, w$ emit the same decisions on the shared suffix, so the two traces would be accepted or rejected together—a contradiction. Hence $c_0, \ldots, c_N$ are pairwise distinct: at least $N+1$ configurations, and with a single location at least $N+1$ register valuations, which no domain of fixed finite size accommodates for all N. □

7. Conclusions, related and future work

We have sketched a minimal language of policies with effects and the deterministic, explicitly stateful automaton model it induces, i.e., *policy automata*. The model is well-behaved: well-formed policies induce a deterministic monitor (Proposition 3.3) recognising a safety property (Proposition 3.7). It is also analysable: over a decidable guard logic, well-formedness (Proposition 3.4), reachability of a denial (Proposition 4.2), and, for finitary automata, policy subsumption and equivalence (Proposition 4.4) are all decidable. It is also compositional: compliant languages are closed under conjunction (Proposition 5.1) and disjunction (Proposition 5.2), with deny-overriding rule-set union recovering the conjunction (Proposition 5.4). And the state earns its place: effects make policy automata strictly more expressive than the stateless Cedar/Rego core (Proposition 6.2), with a memory requirement that provably grows with the usage bound (Proposition 6.3).

Related work. Policy automata sit squarely in the lineage of automata with explicit state variables. They are a deterministic instance of *extended finite state machines* [5], with the policy effect playing the role of the transition's assignment. When guards compare the event parameters against the registers, they are *symbolic register automata* [4], the symbolic refinement of register automata [3], augmented with a Mealy-style decision output in place of plain acceptance; *timed automata* [11] are a further special case of real-valued clock variables. For the monitoring use case, they are close in spirit to *quantified event automata* [6] and the *edit automata* [12], although this one modifies the traces in order to respect the desired safety property. Against usage automata [1], the distinguishing choices are the *explicit, mutable state* carried by the registers (rather than abstract control states reached by instantiation) and *determinism*. Table 1 summarises the comparison.

The term *policy automata* has been used before in the literature; e.g. in [13] it is a synonym for a non-monotonic logic of defeasible rules: there each automaton *votes* on a transaction through defeasible-logic rules, and the verdict of a collection of them is resolved in that non-monotonic logic, which may also report a conflict. Combination is thus external to the model, whereas ours is internal.

Table 1
Policy automata against similar data-aware automata models. "State" is what the automaton mutates beyond the control location; "symbolic guards" are over a decidable logic of values, not merely equality of input against stored values; "effects" are generic state transformers as opposed to plain input-storage. The right column is how a verdict is communicated. Policy automata are the all-yes row, the combination tailored to policy enforcement.

Model	State	Symbolic guards	Effects	Deterministic	Verdict
Usage automata [1]	control only	equality	—	—	bad-state
Register automata [3]	registers	equality	input-storage	det. fragment	final-state
Symbolic reg. automata [4]	registers	yes	input-storage	det. fragment	final-state
EFSM [5]	variables	yes	yes	yes	Mealy
QEA [6]	slice bindings	limited	—	per slice	per-slice
Policy automata	**registers**	**yes**	**yes**	**yes**	**Mealy decision**

On the policy-language side, several proposals enrich stateless authorization with state. In [14], the authors share our motivation (keeping state updates out of the resource guard and inside the policy) but their formalism is a Datalog-like logic whose state is a set of *facts* inserted and retracted by requests, with semantics given in Transaction Logic and a proof system for reasoning about request sequences. Closest to our formulation is *Strobilus* [10] already cited in the introduction: a lightweight imperative DSL of *obligations* on Cedar's entity store, executed *after* a stateless Cedar evaluation has issued its verdict. Policy automata fuse decision and update into a single transition: the Mealy reading that exposes the automaton-theoretic structure underlying such extensions and supplies the metatheory (Propositions 3.4, 4.2, 4.4 and 5.4) that an implementation-oriented language needs but does not provide on its own. The analogous extension of Rego/OPA, *OWSM* [9], maintains a separate state store consulted during evaluation, thus the verdict logic itself remains stateless and effects sit outside the formal core.

At a more conceptual level, the UCON_{ABC} model [15] introduced *attribute mutability* alongside authorizations, obligations, and conditions as a unifying frame for usage control; policy automata can be read as a concrete operational realization of attribute mutability with a deterministic-monitor reading and the decidable analyses developed above.

Future work. The decidability results target the finitary case; the infinite-domain case rests on emptiness and inclusion of the underlying symbolic register automata [4], which are decidable only for restricted theories (Remark 4.5), and mapping out exactly which guard and effect theories stay feasible is the main question left open. Beyond it: pinning down the expressive power of policy automata relative to register [3] and symbolic register automata [4], and to usage [1] and gate automata [16]; giving a faithful, effect-based encoding of Cedar [7] and Rego [8] fragments; and connecting the monitoring view with quantified event automata [6]. Moreover, the pattern instantiated here, reading an event-driven rule language as a class of automata with explicit state, is not specific to access control, and we expect it to apply to *AbU* [17], a calculus of attribute-based memory updates whose event-condition-action rules fire on local memory changes and act on the memory of every node satisfying an attribute condition.

A further direction is policy *synthesis*, the converse of the run we study here. By Proposition 3.5 every automaton is, up to a location register, a flat rule set, so recovering rules from an automaton is immediate. It would be more interesting starting from a *specification* rather than an automaton: given a safety property to guarantee, synthesise the *least restrictive* policy that enforces it (i.e., the maximally permissive supervisor of Ramadge-Wonham control [18]). Given instead a sample of labelled traces (compliant/violating, or request/decision pairs), one would *learn* a minimal consistent policy automaton, extending active automata learning [19] over registers [20] and symbolic guards [21], and the access-control *policy mining* tradition [22], from stateless rules to genuinely stateful usage policies. Our metatheory already supplies the ingredients such procedures need: deciding equivalence (Proposition 4.4) is the verification (and L^* equivalence) oracle, and the Myhill-Nerode argument behind Proposition 6.3 pins down a canonical minimal target. The hardness lies elsewhere (inferring the state variables, guards, and updates over an infinite data domain, where even the stateless minimal-automaton problem is intractable) so realistic results will be query-based or heuristic.

Declaration on Generative AI

During the preparation of this work, the authors used Claude Opus 4.8 in order to: Grammar and spelling check, Drafting content, Peer review simulation. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Bartoletti, Usage automata, in: Foundations and Applications of Security Analysis (ARSPA-WITS 2009), Revised Selected Papers, volume 5511 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 52–69. doi:10.1007/978-3-642-03459-6_4.

[2] M. Bartoletti, P. Degano, G. L. Ferrari, R. Zunino, Local policies for resource usage analysis, ACM Transactions on Programming Languages and Systems 31 (2009) 23:1–23:43. doi:10.1145/1552309.1552313.

[3] M. Kaminski, N. Francez, Finite-memory automata, Theoretical Computer Science 134 (1994) 329–363. doi:10.1016/0304-3975(94)90242-9.

[4] L. D'Antoni, T. Ferreira, M. Sammartino, A. Silva, Symbolic register automata, in: Computer Aided Verification, volume 11561 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 3–21. doi:10.1007/978-3-030-25540-4_1.

[5] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines—a survey, Proceedings of the IEEE 84 (1996) 1090–1123. doi:10.1109/5.533956.

[6] H. Barringer, Y. Falcone, K. Havelund, G. Reger, D. E. Rydeheard, Quantified event automata: Towards expressive and efficient runtime monitors, in: FM 2012: Formal Methods, volume 7436 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 68–84. doi:10.1007/978-3-642-32759-9_9.

[7] J. W. Cutler, C. Disselkoen, A. Eline, S. He, K. Headley, M. Hicks, K. Hietala, E. Ioannidis, J. Kastner, A. Mamat, D. McAdams, M. McCutchen, N. Rungta, E. Torlak, A. M. Wells, Cedar: A new language for expressive, fast, safe, and analyzable authorization, Proc. ACM Program. Lang. 8 (2024). doi:10.1145/3649835.

[8] Open Policy Agent, Open policy agent: Policy language (rego), https://www.openpolicyagent.org/docs/latest/policy-language/, 2026. Accessed May 2026.

[9] M. Baldo, F. I. Ion, M. Miculan, M. Paier, V. Riccio, OWSM: Empowering Rego for stateful access control, in: Proceedings of the Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), volume 3962 of *CEUR Workshop Proceedings*, 2025. URL: https://ceur-ws.org/Vol-3962/paper49.pdf.

[10] M. Baldo, P. Di Gianantonio, M. Miculan, M. Paier, Strobilus: Enriching Cedar with stateful policies, in: Proceedings of the 31st ACM Symposium on Access Control Models and Technologies (SACMAT 2026), ACM, 2026. doi:10.1145/3750555.3811892.

[11] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126 (1994) 183–235. doi:10.1016/0304-3975(94)90010-8.

[12] J. Ligatti, L. Bauer, D. Walker, Edit automata: enforcement mechanisms for run-time security policies, Int. J. Inf. Secur. 4 (2005) 2–16. doi:10.1007/s10207-004-0046-8.

[13] M. McDougall, R. Alur, C. A. Gunter, A model-based approach to integrating security policies for embedded devices, in: Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT 2004), ACM, 2004, pp. 211–219. doi:10.1145/1017753.1017789.

[14] M. Y. Becker, S. Nanz, A logic for state-modifying authorization policies, in: Computer Security (ESORICS 2007), volume 4734 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 203–218. doi:10.1007/978-3-540-74835-9_14.

[15] J. Park, R. S. Sandhu, The UCON$_{abc}$ usage control model, ACM Transactions on Information and System Security 7 (2004) 128–174. doi:10.1145/984334.984339.

[16] G. Costa, I. Matteucci, Trust-driven policy enforcement through gate automata, in: Fifth Inter-

national Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS 2011), IEEE Computer Society, 2011, pp. 208–215. doi:10.1109/IMIS.2011.88.

[17] M. Miculan, M. Pasqua, A calculus for attribute-based memory updates, in: Theoretical Aspects of Computing (ICTAC 2021), volume 12819 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 366–385. doi:10.1007/978-3-030-85315-0_21.

[18] P. J. G. Ramadge, W. M. Wonham, The control of discrete event systems, Proceedings of the IEEE 77 (1989) 81–98. doi:10.1109/5.21072.

[19] D. Angluin, Learning regular sets from queries and counterexamples, Information and Computation 75 (1987) 87–106. doi:10.1016/0890-5401(87)90052-6.

[20] S. Cassel, F. Howar, B. Jonsson, B. Steffen, Active learning for extended finite state machines, Formal Aspects of Computing 28 (2016) 233–263. doi:10.1007/s00165-016-0355-5.

[21] S. Drews, L. D'Antoni, Learning symbolic automata, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017), Part I, volume 10205 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 173–189. doi:10.1007/978-3-662-54577-5_10.

[22] Z. Xu, S. D. Stoller, Mining attribute-based access control policies, IEEE Transactions on Dependable and Secure Computing 12 (2015) 533–545. doi:10.1109/TDSC.2014.2369048.