International Journal of Innovative Computing, Information and Control
Volume **22**, Number **2**, April **2026**

ICIC International ©2026 ISSN 1349-4198
pp. 431–443

REGO2EVENT-B: FORMAL ANALYSIS OF REGO ACCESS CONTROL POLICIES USING EVENT-B

Thanh-Binh Trinh[1], Ngoc-Minh Le[2] and Nguyen Viet Ha[3]

[1]Faculty of Information Systems
Phenikaa University
Nguyen Van Trac Street, Ha Dong, Hanoi 150000, Vietnam
binh.trinhthanh@phenikaa-uni.edu.vn

[2]Faculty of Information Technology
Haiphong University
171 Phan Dang Luu Street, Hai Phong 180000, Vietnam
minhln@dhhp.edu.vn

[3]Institute for Artificial Intelligence
VNU University of Engineering and Technology
144 Xuan Thuy Street, Hanoi 100000, Vietnam
hanv@vnu.edu.vn

Received September 2025; revised December 2025

Abstract. *Rego, the policy language of Open Policy Agent (OPA), is widely used in cloud-native environments for Kubernetes admission control, application programming interface gateway filtering, and continuous integration/continuous delivery compliance. However, ensuring the correctness of complex Rego policies remains challenging, motivating the need for more rigorous analysis and verification techniques. This paper presents Rego2Event-B, a framework that translates Rego policies into semantically equivalent Event-B models for proof-based verification with the Rodin platform. Evaluation on the Rule-Based Traffic Controller case study demonstrates semantic preservation and a 100% proof-obligation discharge rate across 32 generated proof obligations, covering priority, fairness, conflict prevention, and liveness. This work addresses a critical gap in access control verification and promotes the integration of formal methods into security policy engineering.*

Keywords: Open Policy Agent, Rego, Kubernetes, Event-B, Rodin, Rego2Event-B

1. **Introduction.** Access control policies are a foundation of modern security architectures, determining which subjects are authorized to perform which actions on specific resources under defined conditions [1, 2]. In recent years, the shift toward cloud-native and microservices-based infrastructures has driven the adoption of *policy-as-code* paradigms, in which policies are expressed in high-level domain-specific languages (DSLs) and evaluated at runtime by dedicated policy engines.

Rego, the language of OPA [3], powers policy-as-code in cloud-native systems. Its declarative syntax, rich data model, and flexible evaluation semantics make it suitable for expressing complex access control logic and contextual constraints. However, as Rego policies grow in size, complexity, and dynamism, ensuring their correctness, consistency, and safety becomes increasingly challenging. In safety- and security-critical environments, undetected policy flaws – such as conflicts, redundancies, deadlocks, livelocks, or fairness violations – can lead to system failures, security breaches, or violations of regulatory requirements [4]. At present, the primary assurance techniques for Rego policies rely on

DOI: 10.24507/ijicic.22.02.431

testing and simulation, which, while useful, cannot guarantee the absence of subtle logical errors across all possible states and inputs [5, 6]. In particular, testing may fail to uncover potential conflicts, policy redundancies, or violations of liveness and fairness properties. In addition, existing techniques do not provide formal guarantees that all access control requirements are correctly enforced under every possible scenario. This limitation highlights a clear gap in the current state of Rego policy assurance: there is no systematic, proof-based method to formally verify the correctness and consistency of policies.

To address this gap, we present Rego2Event-B, a formal framework that translates Rego policies into semantically equivalent Event-B models for proof-based verification using the Rodin platform [7]. By defining systematic mapping rules from Rego constructs – such as packages, rules, and comprehensions – to Event-B contexts and machines, our framework enables the automated generation of models that can be formally proven to satisfy critical access control properties. We evaluate Rego2Event-B on a representative case study, the Rule-Based Traffic Controller, which captures typical challenges in access control: priority handling, fairness enforcement, deadlock/livelock avoidance, and conflict prevention. The results show semantic preservation, high proof obligation discharge rates, and identification of Rego features requiring preprocessing.

The remainder of this paper is structured as follows. Section 2 introduces the background on Rego, OPA, and Event-B. Section 3 details the proposed `Rego2Event-B` framework. Section 4 presents the traffic controller case study, experimental results, and discusses limitations. Section 5 reviews related work in policy verification, and Section 6 summarizes the findings and outlines directions for future research.

2. Preliminaries.

2.1. Rego and the Open Policy Agent (OPA).

Rego [3, 8, 9] is the policy language of the *Open Policy Agent* (OPA), a widely adopted open-source engine for enforcing access control and compliance policies in cloud-native environments. A Rego policy consists of one or more *packages*, which contain *rules* that define authorization logic. Rules may include *queries*, *comprehensions*, *sets/maps*, and built-in predicates. The evaluation of rules produces *decisions* such as `allow` or `deny` for a given request. While Rego supports rich logical constructs, certain combinations of rules and data may result in *potential conflicts* or ambiguities that are not easily detected at runtime through conventional testing or simulation. For instance, Listing 1 illustrates a simplified Rego policy for a 4-way traffic intersection. Rule `green_NS` allows North-South traffic to go if East-West is red, whereas `green_EW` allows East-West traffic if North-South is red. Requests where both directions have waiting cars satisfy the conditions of both rules, creating a potential conflict in decision outcomes.

Listing 1. Potential conflicting rules in a Rego policy for a 4-way traffic intersection

```
package traffic.intersection
green_NS {
    input.NS.cars > 0
    input.EW.cars == 0
}
green_EW {
    input.EW.cars > 0
    input.NS.cars == 0
}
```

2.2. **Event-B and the Rodin platform.** Event-B [10] is a state-based formal modeling method designed to support correct-by-construction development of reliable and high-assurance systems. It builds upon the classical B-method [11], emphasizing system correctness through abstraction, refinement, and mathematical proofs. Event-B models systems as dynamic collections of variables and events constrained by invariants that must hold at all times.

An Event-B model consists of two primary components: *contexts* and *machines*. Contexts capture the static part of a specification, such as carrier sets, constants, and axioms. Machines describe the dynamic part, including variables, invariants, and events that modify the state. Contexts can be seen by machines, enabling modularity and reuse. The generic structures of contexts and machines are illustrated in Figure 1.

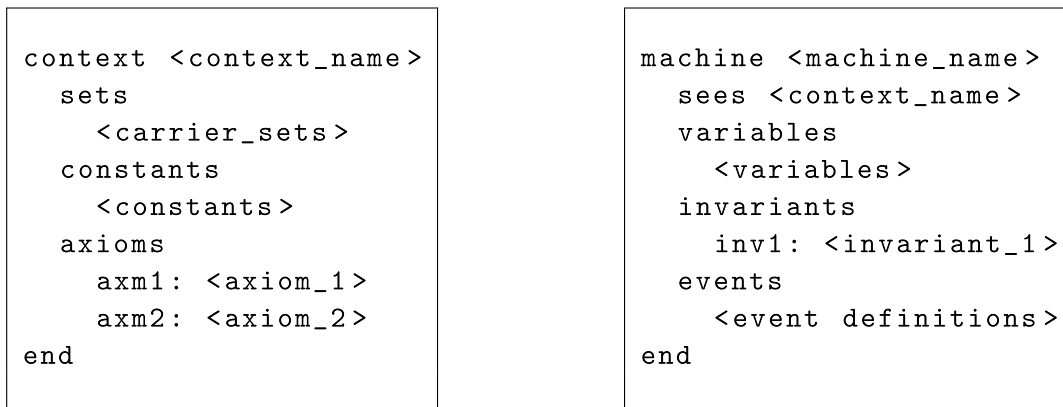


FIGURE 1. Generic structures of context and machine in Event-B

In a *machine*, *events* define system transitions using a guarded command structure. A generic Event-B event can be written compactly as <event_name>≜event(<event_type>) any <parameters> where <guards> then <actions> end. An event is enabled when its guards hold, and its actions update the system state accordingly. Machines support *ordinary events* (abstract behavior) and *refined events* (concrete implementations preserving correctness).

The Rodin platform [7] provides an integrated environment for modeling, type checking, proof obligation generation, and automated or interactive proofs. In the context of access control verification, Event-B enables formal modeling of policy rules, evaluation conditions, and decision points as events and invariants.

3. The Rego2Event-B Framework.

3.1. **Formal definition of Rego policies.** We formally define the core constructs of Rego to enable a transformation to Event-B. This definition highlights the essential components of Rego and can be applied across various domains.

Definition 3.1 (Input/Attribute). *An* input *represents contextual information required for policy evaluation. Each input attribute is a key-value pair:*

$$\mathcal{ATTR} \triangleq \langle id, val, type \rangle$$

where:

- $id \in \mathcal{ID}$ *uniquely identifies the attribute,*
- $val \in \mathcal{VAL}$ *is the attribute value,*
- $type \in \mathcal{TYPE}$ *specifies the data type (boolean, integer, enumeration, etc.).*

Example 3.1 (Traffic Controller Input). *An attribute such as ⟨lane_N_emergency, TRUE, boolean⟩ indicates the presence of an emergency vehicle in the North lane. This input can trigger priority handling rules.*

Definition 3.2 (Rule). *A* rule *defines a conditional decision over input attributes. It consists of*

- *a* head*, specifying the inferred decision or action,*
- *a* body*, a set of predicates over input attributes that must all hold for the head to be inferred.*

Formally: $Rule \triangleq \langle RuleID, Head, \{Predicate_i\}_{i=1}^{n} \rangle$

Example 3.2 (Traffic Light Rule). `allow_emergency_N` *permits the North lane light to turn GREEN if an emergency vehicle is present:*

- *Head: GREEN_N*
- *Body predicates: lane_N_emergency = TRUE*

Definition 3.3 (Package). *A* package *groups related rules under a namespace:*

$$Package \triangleq \langle PackageID, \{Rule_i\}_{i=1}^{n} \rangle$$

Example 3.3 (Traffic Controller Package). *Package* `traffic.controller.rules` *contains rules for lane priority, conflict prevention, and pedestrian crossing. For instance:*

{`allow_emergency_N`, `allow_N_S_GREEN`, `pedestrian_cross_W`}

Definition 3.4 (Rule Conflict). *Two rules* $r_1, r_2 \in \mathcal{R}$ *are in conflict if there exists an input req such that*

- *All predicates in* r_1 *and* r_2 *are satisfied by req,*
- *Their heads specify mutually exclusive or inconsistent decisions.*

Example 3.4 (Conflict in Traffic Rules). `allow_N_S_GREEN` *and* `allow_E_W_GREEN` *would conflict if both evaluate to TRUE at the same time, potentially causing a collision. Detecting this conflict is critical before deployment.*

Definition 3.5 (Fairness). *A rule set satisfies* fairness *if all entities (e.g., lanes, and directions) eventually receive service or access:* $\forall e \in Entities, \exists t \in \mathbb{N} : Granted(e, t) = TRUE$.

Example 3.5 (Fairness in Traffic). *Even if North lane always has emergency vehicles, the system must eventually grant GREEN to East, South, and West lanes to prevent starvation.*

Definition 3.6 (Deadlock and Livelock Avoidance). *A rule set avoids deadlocks if at least one action is always enabled:* $\exists a \in Actions : Enabled(a, t) = TRUE$. *It avoids livelocks if every entity eventually progresses, even under continuously changing inputs.*

Example 3.6 (Deadlock/Livelock in Traffic). *At no point should all lanes be RED indefinitely (deadlock), nor should the GREEN light keep switching between two lanes endlessly without allowing others to proceed (livelock).*

Definition 3.7 (Derived Rule/Comprehension). *Rego allows derived rules or comprehensions that compute sets of decisions from other attributes:* $Derived = \{x | x \in Universe \wedge \phi(x)\}$ *where* ϕ *is a boolean predicate over inputs.*

Example 3.7 (Derived Rule in Traffic). *A rule computes* `allowed_lanes` $=$ *emergency_vehicle(L) = TRUE to give priority dynamically to lanes with emergency vehicles.*

3.2. **Rego to Event-B transformation.** Building upon the formal definitions in Section 2, we adopt a *stepwise refinement approach*, following the Event-B methodology, to formally model and verify Rego access control policies. Figure 2 illustrates the refinement structure, including the contexts, machines, and the incremental modeling steps described below.

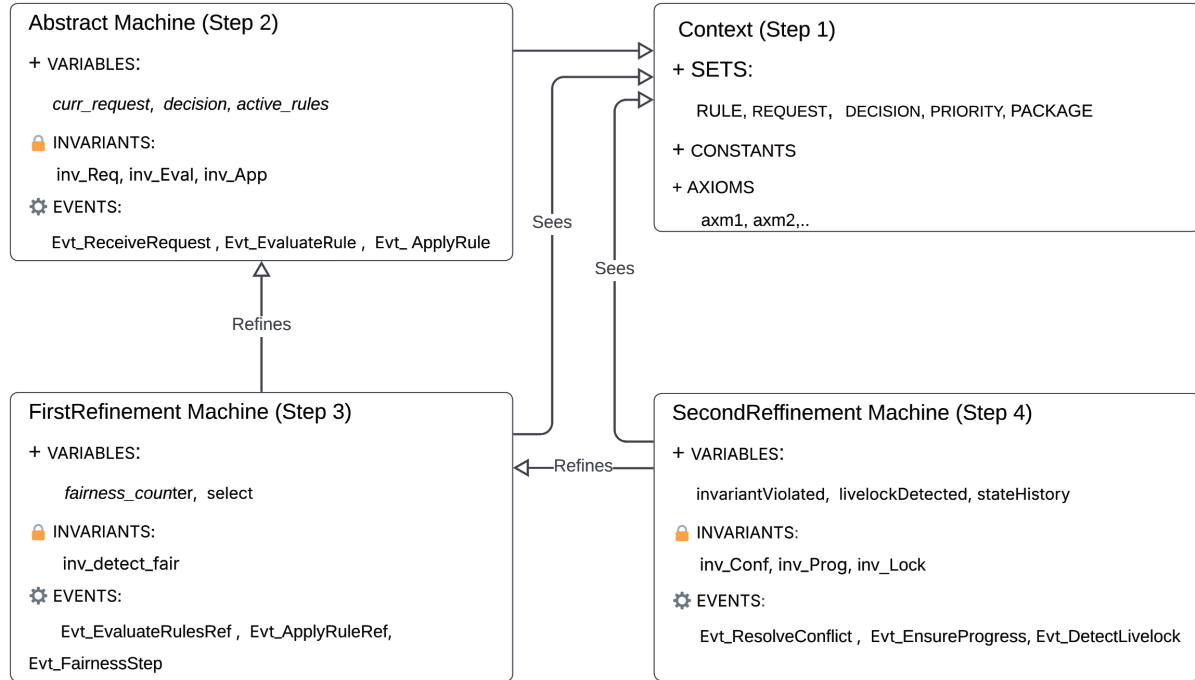
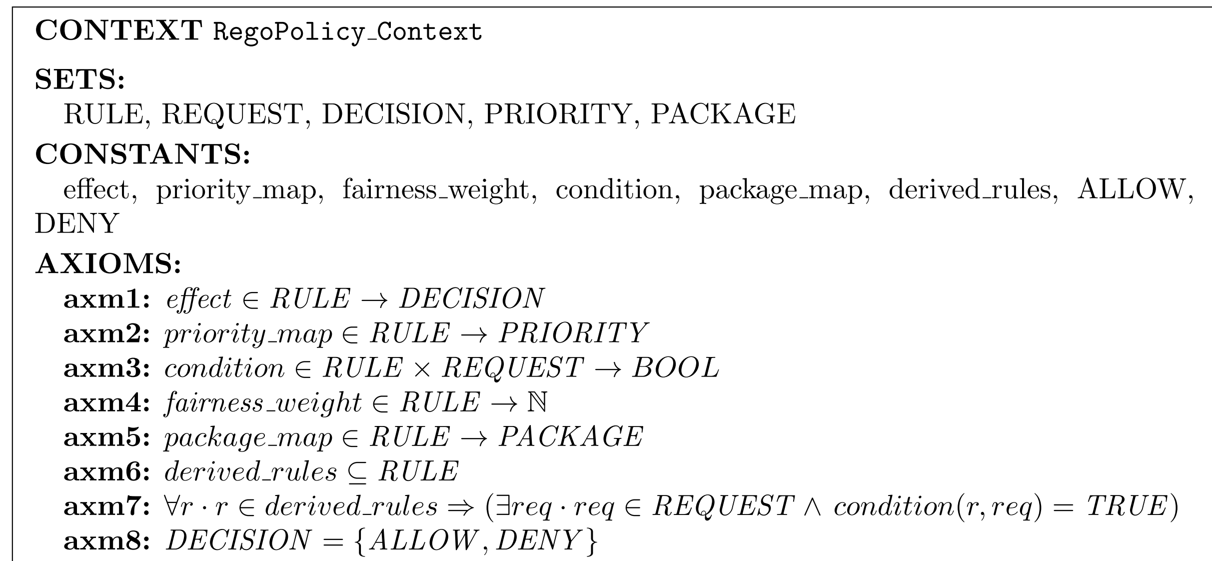


FIGURE 2. Refinement-based modeling structure of Rego using Event-B

Step 1: Modeling the context (static structure). We define the static elements of Rego policies as an Event-B context, as shown in Figure 3. In this context, policy entities are represented as sets `RULE`, `REQUEST`, `DECISION`, `PRIORITY`, and `PACKAGE`, where `RULE` denotes the set of policy rules, `REQUEST` captures possible access requests, `PRIORITY`



CONTEXT `RegoPolicy_Context`

SETS:

RULE, REQUEST, DECISION, PRIORITY, PACKAGE

CONSTANTS:

effect, priority_map, fairness_weight, condition, package_map, derived_rules, ALLOW, DENY

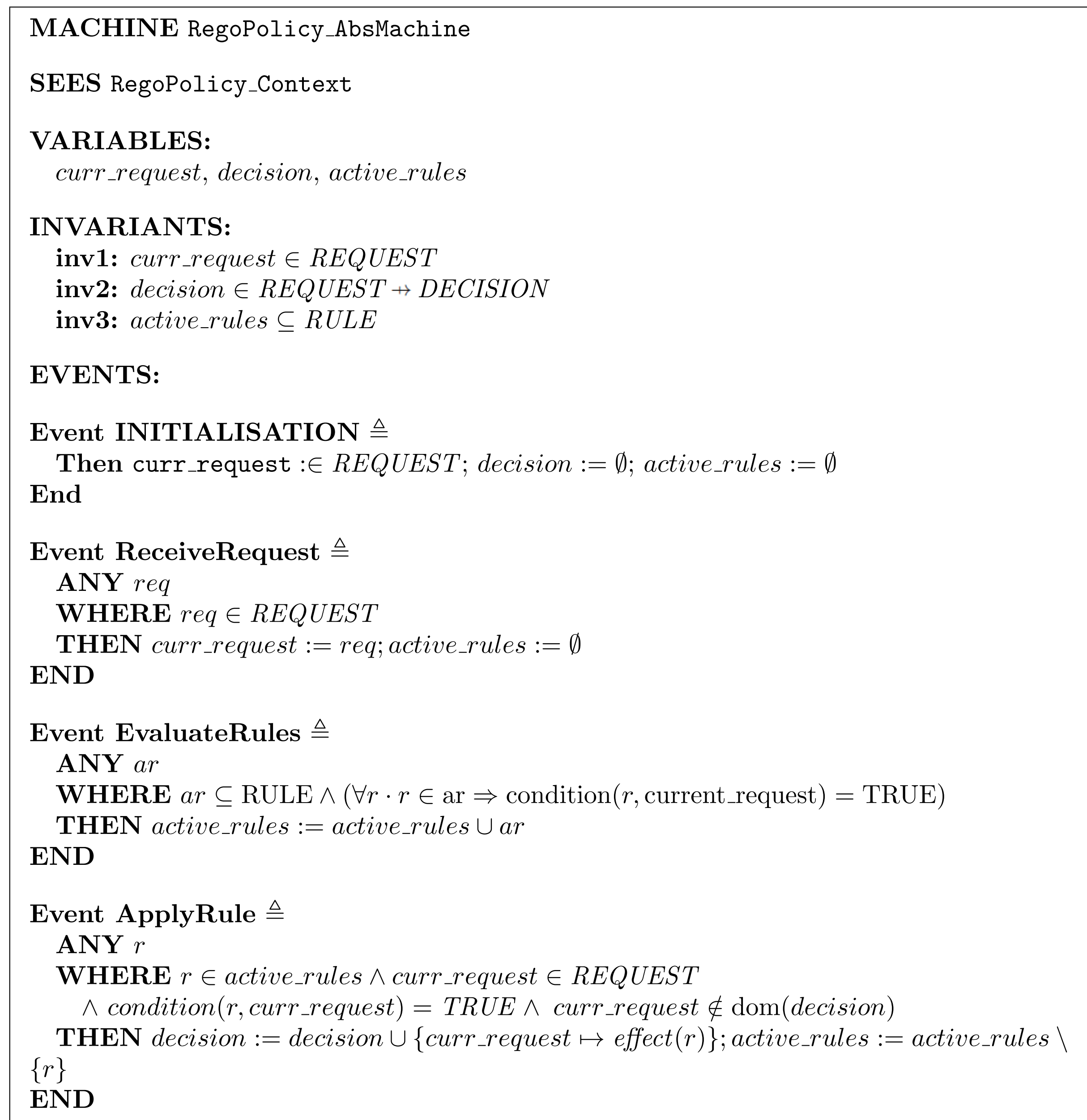
AXIOMS:

axm1: $effect \in RULE \rightarrow DECISION$
axm2: $priority_map \in RULE \rightarrow PRIORITY$
axm3: $condition \in RULE \times REQUEST \rightarrow BOOL$
axm4: $fairness_weight \in RULE \rightarrow \mathbb{N}$
axm5: $package_map \in RULE \rightarrow PACKAGE$
axm6: $derived_rules \subseteq RULE$
axm7: $\forall r \cdot r \in derived_rules \Rightarrow (\exists req \cdot req \in REQUEST \land condition(r, req) = TRUE)$
axm8: $DECISION = \{ALLOW, DENY\}$

FIGURE 3. Event-B context for generic Rego policies

defines the ordering of rules, and PACKAGE groups related rules under a namespace. In addition, axioms (axm1-axm8) formalize these mappings and constraints, ensuring consistent associations for decisions, priorities, conditions, fairness weights, and packages, while also identifying derived rules and restricting the decision set to ALLOW and DENY. This context provides the basis for the abstract machine that models the dynamic behavior of Rego policies.

Step 2: Modeling the abstract machine (dynamic behavior). The abstract machine in Figure 4 models the dynamic evaluation of Rego rules for a given request. The system state is represented by the variables curr_request, decision, and active_rules, and is initialized with an empty decision set and no active rules. The typing invariants ensure that the current request always belongs to the set REQUEST, that decision is a partial function from requests to decisions, and that active_rules is a subset of RULE. Upon



MACHINE RegoPolicy_AbsMachine

SEES RegoPolicy_Context

VARIABLES:
$curr_request$, $decision$, $active_rules$

INVARIANTS:
inv1: $curr_request \in REQUEST$
inv2: $decision \in REQUEST \rightarrow\!\!\!\!\!\!+\; DECISION$
inv3: $active_rules \subseteq RULE$

EVENTS:

Event INITIALISATION $\triangleq$
Then curr_request $:\in REQUEST$; $decision := \emptyset$; $active_rules := \emptyset$
End

Event ReceiveRequest $\triangleq$
ANY req
WHERE $req \in REQUEST$
THEN $curr_request := req; active_rules := \emptyset$
END

Event EvaluateRules $\triangleq$
ANY ar
WHERE $ar \subseteq \text{RULE} \wedge (\forall r \cdot r \in \text{ar} \Rightarrow \text{condition}(r, \text{current_request}) = \text{TRUE})$
THEN $active_rules := active_rules \cup ar$
END

Event ApplyRule $\triangleq$
ANY r
WHERE $r \in active_rules \wedge curr_request \in REQUEST$
$\wedge\; condition(r, curr_request) = TRUE \wedge\; curr_request \notin \text{dom}(decision)$
THEN $decision := decision \cup \{curr_request \mapsto effect(r)\}; active_rules := active_rules \setminus \{r\}$
END

FIGURE 4. Event-B abstract machine for generic Rego policies

receiving a new request, `ReceiveRequest` updates the current request and resets the active rules. Next, `EvaluateRules` nondeterministically activates a subset of rules whose conditions hold for the request. Finally, `ApplyRule` determines the applicability of an active rule and, if satisfied, assigns the corresponding decision while removing the rule from the active set.

This abstract machine provides the foundation for subsequent refinements incorporating priorities, fairness, and conflict resolution.

Step 3: First refinement: priority handling and fairness enforcement. Building upon the abstract machine, this refinement introduces mechanisms to enforce priority and fairness, as illustrated in Figure 5. The system state is extended with two additional variables: `fairness_counter`, a function mapping each rule to a natural number tracking how long it has been overlooked, and `selected`, denoting the currently chosen rule for application. The invariants guarantee that these variables are well-typed and that priority ordering is respected: `fairness_counter` is a total function from `RULE` to $\mathbb{N}$ (inv4), `selected` is always a valid rule (inv5), and lower-priority rules cannot be selected when a higher-priority one is available (inv6).

INVARIANTS

inv4: $fairness_counter \in \text{RULE} \rightarrow \mathbb{N}$
inv5: $selected \in RULE$
inv6: $\forall r_1, r_2 \cdot r_1 \in active_rules \land r_2 \in active_rules \land priority_map(r_1) > priority_map(r_2) \Rightarrow selected \neq r_2$

REFINED EVENTS

Event EvaluateRules_Ref refines $\langle$`EvaluateRules`$\rangle \triangleq$
ANY r
WHERE
$r \in \text{RULE} \land \text{condition}(r, \text{current_request}) = \text{TRUE}$
THEN
$active_rules := active_rules \cup \{r\}; fairness_counter := fairness_counter \oplus \{r \mapsto fairness_counter(r) + 1\}$
END

Event ApplyRule_Ref refines $\langle$`ApplyRule`$\rangle \triangleq$
ANY r
WHERE $r \in active_rules \land \forall r_2 \cdot r_2 \in active_rules \Rightarrow priority_map(r) \geq priority_map(r_2) \land curr_request \notin \text{dom}(decision)$
THEN $selected := r; decision := decision \cup \{curr_request \mapsto \text{effect}(r)\}; active_rules := active_rules \setminus \{r\}; fairness_counter := fairness_counter \oplus \{r \mapsto 0\}$
END

Event FairnessStep $\langle$new event$\rangle \triangleq$
ANY r
WHERE $r \in active_rules \land fairness_counter(r) \geq threshold(r) \land curr_request \notin \text{dom}\,(decision)$
THEN $selected := r; decision := decision \cup \{curr_request \mapsto \text{effect}(r)\}; active_rules := active_rules \setminus \{r\}; fairness_counter := fairness_counter \oplus \{r \mapsto 0\}$
END

FIGURE 5. Refinement for priority handling and fairness enforcement

The refined event `EvaluateRules_Ref` extends rule evaluation by collecting all enabled rules and incrementing their fairness counters. `ApplyRule_Ref` ensures deterministic priority enforcement by selecting the rule with the highest priority and applying its effect,

while also resetting its fairness counter. Finally, `FairnessStep` addresses starvation by forcibly selecting a rule whose counter exceeds its threshold, thereby guaranteeing eventual fairness. Together, these refinements ensure strict priority handling without compromising fairness, and provide the basis for further extensions that address deadlock, livelock, and conflict prevention.

Step 4: Second refinement: deadlock/livelock avoidance and conflict prevention. In this refinement, the machine is extended with mechanisms for deadlock and livelock avoidance as well as conflict prevention, as illustrated in Figure 6. The system state is augmented with the variables `enabled_rules`, representing the currently applicable rules for a request, `applied_rules`, collecting the rules that have already been executed, `conflict_set`, encoding pairs of mutually conflicting rules, and `livelock_counter`, which monitors repeated rule activation.

INVARIANTS

`inv5:` $\texttt{active_req} \neq \emptyset \Rightarrow \texttt{enabled_rules} \neq \emptyset$ (progress guarantee)

`inv6:` $\forall r \cdot r \in \mathit{enabled_rules} \Rightarrow \mathit{livelock_counter}(r) < \mathit{THRESHOLD}$ (livelock freedom)

`inv7:` $\forall r_i, r_j \cdot (r_i, r_j) \in \mathit{conflict_set} \Rightarrow \neg(r_i \in \mathit{applied_rules} \wedge r_j \in \mathit{applied_rules})$ (conflict prevention)

NEW EVENTS

Event ResolveConflict

ANY r_i, r_j

WHERE

$r_i \in \mathit{enabled_rules} \wedge r_j \in \mathit{enabled_rules} \wedge (r_i, r_j) \in \mathit{conflict_set}$

$\mathit{priority_map}(r_i) \geq \mathit{priority_map}(r_j)$

THEN

$\mathit{enabled_rules} := \mathit{enabled_rules} \setminus \{r_j\}; \mathit{applied_rules} := \mathit{applied_rules} \cup \{r_i\}; \mathit{livelock_counter}(r_i) := \emptyset$

END

Event EnsureProgress

WHEN

$\mathit{enabled_rules} = \emptyset \wedge \mathit{active_req} \neq \emptyset$

THEN

$\mathit{decision} := \mathit{DENY}; \mathit{active_req} := \emptyset; \mathit{livelock_counter} := \mathit{RULE_ID} \times \{0\}$

END

Event DetectLivelock

ANY r

WHERE

$r \in \mathit{RULE} \wedge \mathit{livelock_counter}(r) \geq \mathit{THRESHOLD}$

THEN

$\mathit{enabled_rules} := \mathit{enabled_rules} \setminus \{r\}; \mathit{applied_rules} := \mathit{applied_rules} \cup \{r\}$

$\mathit{livelock_counter}(r) := 0$

END

Figure 6. Second refinement: deadlock/livelock avoidance and conflict prevention

The invariants capture the essential safety and liveness guarantees: `inv5` ensures that whenever there is an active request, some enabled rule is available (progress), `inv6` bounds the counters to prevent unproductive cycles (livelock freedom), and `inv7` excludes the simultaneous application of conflicting rules (conflict prevention). To maintain these properties, three new events refine the execution flow. `ResolveConflict` resolves contention

by removing the lower-priority rule from the enabled set and committing the higher-priority one. `EnsureProgress` terminates deadlock-prone states by denying the request and resetting counters. Finally, `DetectLivelock` eliminates rules that exceed their livelock threshold, forcing their eventual application and resetting their counters. To improve reproducibility and clarify the transformation process, we provide a formalized pseudocode specification of the Rego-to-Event-B mapping in Algorithm 1.

Algorithm 1 Rego to Event-B Transformation

Require: Rego policy P with packages, rules, comprehensions
Ensure: Event-B model with contexts, machines, refined events
1: **Step 1: Contexts** – Define sets: RULE, REQUEST, DECISION, PRIORITY, PACKAGE; constants: effect, condition, priority_map, fairness_weight; condition, package_map, derived_rules, ALLOW, DENY; axioms for typing and constraints
2: **Step 2: Abstract Machine** – Variables: *curr_request*, *decision*, *active_rules*
3: **while** request $req \in REQUEST$ arrives **do**
4: $curr_request := req$; $active_rules := \emptyset$
5: /* *EvaluateRules: nondeterministically activate rules* */
6: choose $ar \subseteq RULE$ such that $\forall r \in ar \cdot condition(r, req) = TRUE$
7: $active_rules := active_rules \cup ar$
8: /* *ApplyRule: apply one enabled rule at a time* */
9: **if** $curr_request \notin dom(decision)$ **then**
10: choose $r \in active_rules$
11: $decision := decision \cup \{curr_request \mapsto effect(r)\}$
12: $active_rules := active_rules \setminus \{r\}$
13: **end if**
14: **end while**
15: **Step 3: Priority&Fairness** – Extend vars: *fairness_counter*, *selected*; update counters
16: Select $r_{\max} := \arg\max_{r \in active_rules} priority_map(r)$; $selected := r_{\max}$; $decision(curr_request) := effect(r_{\max})$
17: Apply fairness: select r if $fairness_counter(r) \geq threshold(r)$
18: **Step 4: Conflict, Deadlock, Livelock** – Extend vars: *enabled_rules*, *applied_rules*, *conflict_set*, *livelock_counter*
19: **for all** $(r_i, r_j) \in conflict_set \cap enabled_rules$ **do** apply higher-priority r; remove lower-priority
20: **end for**
21: **if** $enabled_rules = \emptyset \wedge active_req \neq \emptyset$ **then** $decision := DENY$; reset *livelock_counter*
22: **end if**
23: **for all** $r \in RULE$ with $livelock_counter(r) \geq THRESHOLD$ **do** apply r; reset $livelock_counter(r)$
24: **end for**

To formalize the correctness of the transformation, we provide a brief proof sketch showing that the Event-B model generated by Algorithm 1 faithfully preserves the operational semantics of Rego policies.

Correctness Sketch.

1) **Soundness:** For each rule r, if all predicates in r's body evaluate to true, then the event `ApplyRule` assigns *effect*(r), matching Rego's inference semantics.
2) **Completeness:** All enabled rules in Rego correspond to elements of `active_rules`; no rule is omitted.
3) **Determinism under priorities:** The invariant on `priority_map` ensures that only the highest-priority rule can be selected, reflecting Rego's "maximally specific rule" behavior.

4) **Fairness:** `fairness_counter` and `FairnessStep` guarantee eventual service for all rules (liveness).
5) **Conflict-freedom:** `inv7` ensures that no conflicting pair is simultaneously applied.

4. **Case Study.** Building upon the formally verified Rego-to-Event-B framework presented in Section 3, we now illustrate the practical application of our approach through a concrete case study, the Rule-Based Traffic Controller, showing how the Event-B specifications can be instantiated and analyzed for real-world Rego policies.

4.1. **System description and rule set.** We illustrate the applicability of our approach through a case study on a *Rule-Based Traffic Controller*, a critical infrastructure component that regulates vehicle and pedestrian flows at a four-way intersection using Rego policies. The control logic of the traffic light system is captured by a simplified Rego policy, shown in Table 1, which defines the core behaviors and highlights potential verification issues.

TABLE 1. Simplified Rego policy for traffic light control

ID	Description	Condition (Guard)	Action
R1	Priority for NS	`input.emergency_NS == true`	Set priority for NS
R2	Priority for EW	`input.emergency_EW == true`	Set priority for EW

R3	Green for NS	`input.vehicles_NS > 0`
		`input.vehicles_EW == 0`
		`not input.emergency_EW`
			Set green for NS

R4	Green for EW	`input.vehicles_EW > 0`
		`input.vehicles_NS == 0`
		`not input.emergency_NS`
			Set green for EW

R5	Fair rotation	`not input.emergency_NS`
		`not input.emergency_EW`
		`state.last_served != "NS"`
			Rotate service

Verification challenges: **Conflict**: R3 and R4 may activate simultaneously when both directions have vehicles. **Unfairness**: R5 may starve certain directions under biased scheduling. **Deadlock**: Possible if no rule is enabled. **Livelock**: Possible under cyclic scheduling.

4.2. **Experimental results.** Building upon the generic `Rego2Event-B` framework, the traffic light system was modeled through four refinement steps. The verification with Rodin 3.8 achieved full automation, with all POs discharged.

Priority handling. The refinement enforces deterministic priority ordering. For example, when both `R1` (emergency NS) and `R3` (regular NS) are enabled, only `R1` is applied. All related POs were discharged, confirming that no lower-priority rule can override a higher-priority one.

Fairness guarantee. The `FairnessStep` event ensures bounded waiting for all directions. Invariants show that any active rule with a positive fairness weight will eventually be applied. Verification confirmed that starvation is prevented, even under continuous emergency traffic.

Conflict prevention. The `DetectConflict` mechanism identified the potential conflict between `R3_greenNS` and `R4_greenEW` (Table 2). Resolution was achieved by introducing the priority ordering (`R3 > R4`) and refining guards to enforce mutual exclusion.

Deadlock/livelock freedom. As formalized in Figure 6, invariant `inv5` guarantees deadlock freedom by ensuring that at least one rule is always enabled when a request is active. The `DetectLivelock` event prevents livelock by bounding rotation steps via

Table 2. PO for traffic rule conflict detection

$\forall r_i, r_j \cdot$ $r_i = \texttt{R3_greenNS} \wedge r_j = \texttt{R4_greenEW}$ $\wedge\ \texttt{guard}(r_i) = TRUE \wedge \texttt{guard}(r_j) = TRUE$ $\wedge\ \texttt{action}(r_i) = \texttt{set_green(NS)}$ $\wedge\ \texttt{action}(r_j) = \texttt{set_green(EW)}$ $\vdash$ $\texttt{isConflict}(r_i, r_j) = \texttt{TRUE}$	`DetectConflict /inv2/INV`

Table 3. Verification results for traffic light system in Rodin

Refinement level	POs generated	Discharged (%)
Contexts (static)	5	100%
Abstract machine	8	100%
Priority/Fairness	7	100%
Safety/Liveness	12	100%
Total	32	100%

Note: All proof obligations (POs) were automatically discharged by Rodin without manual intervention.

a `livelock_counter`. All related proof obligations were discharged in `Rodin`, confirming the absence of deadlocks and livelocks.

Verification summary. These results are consolidated in Table 3, which shows that all 32 proof obligations were automatically discharged in Rodin. This confirms that the Rego2Event-B framework effectively verifies key properties-priority, fairness, conflict prevention, and liveness-within the traffic controller case study.

4.3. **Evaluation and discussion.** The case study demonstrates that Rego2Event-B can successfully verify critical properties of Rego policies, with all proof obligations automatically discharged in Rodin. This highlights the potential of integrating formal methods into policy engineering to strengthen assurance beyond testing and simulation. However, the evaluation is limited to a single policy domain (traffic control) and a core subset of Rego constructs, leaving advanced features and scalability to large deployments as future work. These limitations may affect generalizability but also point to promising directions for extending the framework. Preliminary analysis on larger rule sets suggests that the number of generated proof obligations grows roughly linearly with the number of rules, indicating that the transformation and verification process scales proportionally and can handle multiple interacting packages, supporting future applications beyond the traffic control domain.

5. Related Work.

Formal verification of policy languages has gained increasing attention as cloud-native systems adopt policy-as-code frameworks. Assurance of such policies is essential, since errors in authorization logic can undermine security and compliance. Existing approaches can be grouped into three categories.

Static formal methods. Static approaches include SMT solving [6], Alloy [12], TLA+ [13], Event-B [14], and formal methods like Yang et al. [15] for detecting conflicts and inconsistencies in access control policies. These methods provide strong formal guarantees and can detect subtle logical flaws that dynamic techniques may miss. However, they require specialized modeling expertise and are not directly integrated with Rego or OPA. Alloy is restricted by bounded model checking, SMT solving faces scalability challenges, and TLA+ excels in temporal reasoning but lacks native support for policy-as-code.

Dynamic analysis techniques. Dynamic approaches rely on testing, simulation, or runtime monitoring to assess policy behavior in operational environments [16, 17]. These methods can capture behaviors that are difficult to model statically and are easier to apply in practice. However, they lack formal completeness and cannot guarantee correctness across all execution paths. Rare conflicts, deadlocks, or violations of fairness and liveness may go undetected, limiting their effectiveness for full assurance.

Hybrid frameworks. Hybrid solutions attempt to combine static rigor with runtime adaptability, e.g., generating runtime monitors from formal specifications or combining rule-learning with model-based analysis [18, 19]. While these frameworks improve robustness, synchronizing static and dynamic views often introduces additional complexity, overhead, and potential gaps between specification and enforcement.

Positioning of this work. Our contribution differs in three key ways. First, `Rego2-Event-B` is the first framework to systematically map Rego constructs into Event-B, providing a native connection between policy-as-code and formal verification. Second, by leveraging Rodin's theorem proving and refinement support, it enables proof-based verification of both safety and liveness properties, ensuring correctness under all possible scenarios – a guarantee that static, dynamic, or hybrid methods alone cannot provide. Third, unlike prior static analyses (e.g., Alloy or TLA+), `Rego2Event-B` is fully integrated with Rego/OPA, reducing modeling overhead and offering end-to-end assurance from policy specification to verified enforcement.

6. Conclusion.

This paper introduced `Rego2Event-B`, a framework for systematically transforming Rego policies into Event-B models for proof-based verification. By formalizing Rego constructs and mapping them into Event-B, the approach leverages the Rodin platform to automatically discharge proof obligations. A case study on a traffic light controller showed that the framework can verify both safety and liveness properties, with all obligations discharged. This demonstrates the feasibility of applying formal methods to strengthening assurance in policy-based control systems. Future work will extend support to advanced Rego features and evaluate scalability in larger deployments. Compared to Alloy- or TLA+-based approaches, Rego2Event-B directly integrates with Rego and OPA, offering end-to-end correctness guarantees from specification to enforcement.

REFERENCES

[1] R. S Sandhu and P. Samarati, Access control: Principle and practice, *IEEE Communications Magazine*, vol.32, no.9, pp.40-48, 1994.

[2] V. C Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller and K. Scarfone, *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*, Technical Report, National Institute of Standards and Technology, 2014.

[3] Open Policy Agent Project, *Open Policy Agent Documentation*, https://www.openpolicyagent.org/docs/latest/, 2023.

[4] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, Wiley, 2001.

[5] M. H. E. Aouadi, K. Toumi and A. Cavalli, Testing security policies for distributed systems: Vehicular networks as a case study, *arXiv Preprint*, arXiv: 1410.5789, 2014.

[6] A. Margheri, M. Masi, R. Pugliese and F. Tiezzi, A rigorous framework for specification, analysis and enforcement of access control policies, *arXiv Preprint*, arXiv: 1612.09339, 2016.

[7] J. Coleman, C. Jones, I. Oliver, A. Romanovsky and E. Troubitsyna, Rodin (rigorous open development environment for complex systems), *The 5th European Dependable Computing Conference: EDCC-5 Supplementary Volume*, 2005.

[8] V. T. D. Jakkaraju, Adversarial-aware kubernetes admission controllers for real-time threat suppression, *International Journal of Intelligent Systems and Applications in Engineering*, vol.8, no.2, pp.143-151, 2020.

[9] Istio Project, *Istio API Gateway and Policy Enforcement*, https://istio.io/latest/docs/, 2021.

[10] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, 2010.
[11] J.-R. Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, Cambridge, UK, 1996.
[12] B. Abdelkrim, An Alloy-based formalization of ABAC in collaborative and non-collaborative cloud environments, *Studies in Computational Intelligence*, 2024.
[13] H. Cirstea, M. A. Kuppe, B. Loillier and S. Merz, Validating traces of distributed programs against TLA+ specifications, *arXiv Preprint*, arXiv: 2404.16075, 2024.
[14] T.-B. Trinh and N.-T. Truong, Conflict analysis of the Chinese wall security policy model using Event-B, *International Journal of Innovative Computing, Information and Control*, vol.21, no.4, pp.973-986, 2025.
[15] Z. Yang, L. Yin, S. Jin and M. Duan, Towards formal security analysis of decentralized information flow control policies, *International Journal of Innovative Computing, Information and Control*, vol.8, no.11, pp.7969-7981, 2012.
[16] V. C. Hu, D. F. Ferraiolo and D. R. Kuhn, *Assessment of Access Control Systems*, Technical Report NISTIR 7316, National Institute of Standards and Technology, 2006.
[17] E. B. Martin, D. Xu and R. Sandhu, A framework for testing access control policies, *Proc. of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT'07)*, pp.139-148, 2007.
[18] M. Rezvani, R. Krishnan and R. Sandhu, Logic-based verification of access control policies, *Journal of Computer Security*, vol.27, no.6, pp.627-653, 2019.
[19] D. Llamas, E. Fernández-Medina and M. Piattini, Hybrid analysis of access control policies using rule learning and model checking, *Proc. of the 28th ACM Symposium on Access Control Models and Technologies (SACMAT'23)*, pp.131-142, 2023.

Author Biography

Thanh-Binh Trinh received the Ph.D. degree in Software Engineering from VNU University of Engineering and Technology, Vietnam, in 2012. He is currently the Head of the Faculty of Information Systems at Phenikaa University, Vietnam. His research interests include software engineering, formal methods, software verification and validation, and model-driven development.

Ngoc-Minh Le received the M.Sc. degree in Information Technology from VNU University of Engineering and Technology, Vietnam, in 2015. He is currently a Lecturer at the Faculty of Information Technology, Haiphong University, Vietnam. His research interests include software testing, formal methods, and artificial intelligence.

Nguyen Viet Ha received the Doctor of Engineering degree from Takushoku University, Japan, in 2002. He is currently an Associate Professor at the Institute for Artificial Intelligence, VNU University of Engineering and Technology, Vietnam. His research interests include natural language processing, deep learning, and software verification.