

# The Intersection of Policy Software Languages and Formal Methods

---

**Author:** Manus AI

**Date:** January 29, 2026

## Introduction

---

In modern computing, the need to manage access control, security, and operational constraints has led to the rise of **policy software languages**. These specialized languages allow organizations to define and enforce rules across their systems in a consistent and automated manner, a practice commonly known as **Policy as Code**. As these policies become increasingly critical for security and compliance, ensuring their correctness is paramount. This has driven a growing interest in the application of **formal methods**—mathematically rigorous techniques for specifying, developing, and verifying software and hardware systems—to the domain of policy languages. This report explores the landscape of policy software languages and examines the extent to which they leverage formal methods to provide higher assurances of correctness and security.

## The Rise of Policy as Code

---

The Policy as Code movement advocates for treating policies as first-class citizens in the software development lifecycle. Instead of being configured through graphical user interfaces or ad-hoc scripts, policies are defined in a high-level language, stored in version control systems, and deployed through automated pipelines. This approach offers numerous benefits, including improved auditability, repeatability, and scalability.

Several languages have emerged as popular choices for implementing Policy as Code:

- **Rego:** The policy language for the Open Policy Agent (OPA) project, Rego is a declarative language designed for expressing policies over complex hierarchical data structures. It has gained significant traction in the cloud-native ecosystem for use cases like Kubernetes admission control and API authorization [1].
- **Sentinel:** Developed by HashiCorp, Sentinel is a policy as code language that is embedded in many of HashiCorp's enterprise products. It is designed to be more accessible to non-programmers and focuses on enforcing policies across the infrastructure provisioning lifecycle [2].
- **General-Purpose Languages:** While specialized languages are common, many organizations also use general-purpose languages like Python and Go, along with data serialization formats like YAML, to define and manage policies.

## Formal Methods in Policy Verification

---

Formal methods provide a level of assurance that is difficult to achieve through traditional testing alone. By using mathematical logic to reason about the behavior of a system, formal methods can prove the absence of certain classes of errors and verify that a system adheres to its specification. In the context of policy languages, formal verification can be used to answer critical questions such as:

- *“Does this policy allow unauthorized access to sensitive data?”*
- *“Is it possible for a user to gain more permissions than intended?”*
- *“Does this change to a policy have any unintended side effects?”*

Several formal methods are used in the verification of policy languages, each with its own trade-offs between expressiveness and automation:

Formal Method	Description	Automation	Expressiveness	Key Use Cases in Policy Verification
<b>SMT Solvers</b>	Satisfiability Modulo Theories (SMT) solvers check the satisfiability of logical formulas with respect to various background theories (e.g., arithmetic, bit-vectors, strings).	High	Moderate	Automated analysis of access control policies (e.g., AWS Zelkova).
<b>Interactive Theorem Provers</b>	Tools like Coq, Isabelle, and Lean allow for the expression of complex properties and the construction of machine-checked proofs.	Low	High	Certification of policy language semantics and properties (e.g., TEpla).
<b>Model Checking</b>	An automated technique that explores all possible states of a system to check if a given property holds.	High	Moderate	Verification of concurrent systems and protocols.

## Case Studies: Formal Methods in Practice

---

To better understand the application of formal methods to policy languages, we will examine three case studies that showcase different approaches to verification.

### AWS Zelkova: Automated Reasoning at Scale

Amazon Web Services (AWS) has been a pioneer in the use of formal methods for verifying access control policies. **Zelkova** is a policy analysis engine that uses SMT solvers to analyze AWS Identity and Access Management (IAM) policies [3]. Zelkova translates IAM policies into logical formulas and then uses SMT solvers like Z3 and CVC4 to answer questions about the policies. This allows AWS to detect entire classes

of misconfigurations, such as publicly accessible S3 buckets, and to provide customers with tools like IAM Access Analyzer to help them secure their resources. Zelkova is used millions of times a day and can analyze most policies in milliseconds, demonstrating that formal methods can be applied at scale in a real-world setting.

## Cedar: Verification-Guided Development

**Cedar** is a new open-source policy language developed by AWS that was designed from the ground up with formal methods in mind [4]. Cedar follows a process called **verification-guided development**, which involves two key activities:

1. **Formal Modeling in Dafny:** The semantics of Cedar and its authorization engine are modeled in Dafny, a verification-aware programming language. This allows developers to prove critical properties of the language, such as “a deny rule always overrides a permit rule.”
2. **Differential Random Testing:** The production implementation of Cedar, written in Rust, is continuously tested against the Dafny model to ensure that it behaves as expected.

This approach provides a high degree of confidence in the correctness of the Cedar implementation and has already helped to prevent subtle bugs that would have been difficult to find with traditional testing.

## TEpla: A Certified Policy Language

**TEpla** is a certified access control policy language that has been formally verified using the **Coq proof assistant** [5]. Unlike the automated approach of SMT solvers, Coq requires interactive, human-guided proof construction. This allows for the verification of more complex properties and provides a higher level of assurance. The entire semantics of TEpla, along with key security properties, have been formally specified and proven in Coq. This makes TEpla a “certified” language, where the language itself has been mathematically proven to be correct.

## The Spectrum of Formal Methods Adoption

---

The use of formal methods in policy languages exists on a spectrum, from languages with deep, built-in verification to those with more ad-hoc approaches.

Level of Adoption	Examples	Description
<b>Full Formal Verification</b>	Cedar, TEpla	The language is designed for verification, and its semantics and key properties are formally proven.
<b>Automated Reasoning Tools</b>	AWS IAM (with Zelkova)	An existing language is retrofitted with formal verification tools to analyze policies.
<b>Formal Specification</b>	XACML, Ponder	The language has a formal syntax and semantics, but verification is not a primary focus.
<b>Declarative with Logical Foundations</b>	Rego	The language is based on a logical foundation (Datalog), but formal verification is not a standard practice.

The trend in modern policy languages is clearly towards deeper integration of formal methods. By designing languages with verification in mind, developers can create more secure, reliable, and trustworthy policy-driven systems.

## Conclusion

---

Policy software languages are becoming an essential component of modern software systems. As the complexity and importance of these policies grow, so does the need for strong assurances of their correctness. Formal methods provide a powerful set of tools for achieving this assurance. From the large-scale automated reasoning of AWS Zelkova to the verification-guided development of Cedar and the certified approach of TEpla, the use of formal methods in policy languages is a rapidly evolving field. As more organizations adopt Policy as Code, we can expect to see a continued and growing emphasis on the use of formal methods to ensure that these policies are not only powerful but also provably correct.

## References

---

- [1] Open Policy Agent. (n.d.). *Policy Language*. Retrieved from <https://www.openpolicyagent.org/docs/latest/policy-language/>

- [2] HashiCorp. (n.d.). *Sentinel Language*. Retrieved from <https://docs.hashicorp.com/sentinel/concepts/language>
- [3] Backes, J., Bolignano, P., Cook, B., Gacek, A., Kasper, L., NeKyun, O., Rungta, N., & Varming, C. (2018). *Semantic-based Automated Reasoning for AWS Access Policies using SMT*. In *Formal Methods in Computer-Aided Design (FMCAD)*.
- [4] Amazon Science. (2023, May 10). *How we built Cedar with automated reasoning and differential testing*. Retrieved from <https://www.amazon.science/blog/how-we-built-cedar-with-automated-reasoning-and-differential-testing>
- [5] Eaman, A., & Felty, A. (2024). *A certified access control policy language: TEpla*. *International Journal on Software Tools for Technology Transfer*.