ControlsDSL: A Language for Verifiable Cloud Configuration Controls

Miyriee Alair
Amazon Web Services
Los Angeles, USA
mcgalair@amazon.com

Shrutarshi Basu
Amazon Web Services
Santa Clara, USA
basushru@amazon.com

Sean Cai
Amazon Web Services
Los Angeles, USA
caisea@amazon.com

Loris D'Antoni
Amazon Web Services
San Diego, USA
lorisd@amazon.com

Julio Delgado Jr.
Amazon Web Services
Miami, USA
delgjul@amazon.com

Antonio Filieri✉
Amazon Web Services
Santa Clara, USA
afilieri@amazon.com

Andrew Gacek
Amazon Web Services
Minneapolis, USA
gacek@amazon.com

Anand Goyal
Amazon Web Services
Santa Clara, USA
anandgo@amazon.com

Sebastiaan Joosten
Amazon Web Services
Minneapolis, USA
sjoosten@amazon.com

Varad Panch
Amazon Web Services
Santa Clara, USA
varadvp@amazon.com

Sorawee Porncharoenwase
Amazon Web Services
Santa Clara, USA
soraweep@amazon.com

Vishwanath Raman
Amazon Web Services
Santa Clara, USA
mrvraman@amazon.com

Niloofar Razavi
Amazon Web Services
Santa Clara, USA
razavin@amazon.com

Neha Rungta
Amazon Web Services
Santa Clara, USA
rungta@amazon.com

Mike Schliep
Amazon Web Services
Minneapolis, USA
schliepm@amazon.com

Preethi Sekaran
Amazon Web Services
Santa Clara, USA
preesek@amazon.com

Chase Johnson*
University of Minnesota
Minneapolis, USA
me@chasej.dev

Abstract

Cloud providers rely on compliance controls to ensure that customer resources are configured according to security best practices — from encryption of storage to network isolation of compute instances. Today, these controls are typically written as general-purpose programs (e.g., in Python or Java), making them difficult to analyze, test exhaustively, and maintain.

We present `ControlsDSL`, a domain-specific language for writing compliance controls. `ControlsDSL`'s restricted semantics enables analysis based on Satisfiability Modulo Theories (SMT), generating test inputs automatically, verifying semantic equivalence between control versions, and supporting *scenario generation*. Given a control, `ControlsDSL` produces a set of human-readable scenarios that describe every possible execution path and its expected result. We prove that generated scenarios are *sound* (they accurately predict the control's behavior) and *complete* (every well-typed input is covered by at least one scenario).

We evaluate `ControlsDSL` on 397 compliance controls deployed in production at Amazon Web Services (AWS), demonstrating its expressiveness, the effectiveness of scenario generation, and the practical value of SMT-based analysis for control development and migration. `ControlsDSL` controls serve over one billion daily compliance evaluations and are 3.3× more concise than their Python/Java counterparts. Scenario generation completes in under one second for the vast majority of controls, and SMT-based test generation achieves complete path coverage—completing in seconds for the median control, with a heavy tail (minutes) for the more complex controls.

*Work done while interning at Amazon Web Services.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834488

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Security and privacy → Logic and verification**.

Keywords

compliance, cloud configuration, domain-specific language

ACM Reference Format:
Miyriee Alair, Shrutarshi Basu, Sean Cai, Loris D'Antoni, Julio Delgado Jr., Antonio Filieri, Andrew Gacek, Anand Goyal, Sebastiaan Joosten, Varad Panch, Sorawee Porncharoenwase, Vishwanath Raman, Niloofar Razavi, Neha Rungta, Mike Schliep, Preethi Sekaran, and Chase Johnson. 2026. `ControlsDSL`: A Language for Verifiable Cloud Configuration Controls. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834488

1 Introduction

Cloud configuration compliance—the problem of ensuring that cloud resources adhere to organizational security and operational policies—is a central concern for every enterprise cloud deployment, from ensuring S3 buckets are encrypted to verifying that EC2 instances follow network isolation rules.

```python
def check_bucket_compliance(bucket):
  enc = bucket.get('ServerSideEncryptionConfig')
  if not enc:
    return 'NonCompliant'
  pub = bucket.get('PublicAccessBlockConfig')
  if pub and not pub.get('BlockPublicAcls'):
    return 'NonCompliant'
  for tag in bucket['Tags']: # Tags optional; missing throws
    if tag['Key'] == 'Env' and tag['Value'] == 'Prod':
      vc = bucket.get('VersioningConfig', {})
      if vc.get('Status') != 'Enabled':
        return 'NonCompliant'
  return 'Compliant'
```

Figure 1: Embedded compliance logic in Python.

A common solution is to embed compliance logic in imperative code. For example, the `check_bucket_compliance` Python function (Figure 1), inspired by a production compliance check, validates AWS S3 bucket configurations[1]. Directly encoding compliance logic in a general-purpose language has three major drawbacks. First, it is *hard to write correctly*: edge cases like missing nested fields or `null` values are easy to overlook and can lead to security vulnerabilities or false positives that erode trust in the compliance system. Second, embedded controls are *hard to audit*: an auditor must mentally trace all execution paths to verify invariants such as "all production buckets must have versioning enabled." Finally, embedded controls are *hard to test comprehensively*; engineers do not write exhaustive test cases. Inadequate testing means that edge cases—like missing nested fields, empty arrays, or unexpected field combinations—often go untested. When controls change, engineers must manually update test cases and re-verify that each execution path produces the intended result. This manual process is error-prone: important paths are frequently overlooked, and security-critical corner cases may go undetected.

[1]In this paper, we focus on AWS configurations, but the proposed techniques are extensible to most JSON-based configuration systems.

Declarative controls with automatic verification. A better alternative is to express configuration controls as declarative programs in a domain-specific language (DSL), separating the control logic from its evaluation. Declarative controls make compliance logic explicit and easier to audit, but this alone does not solve the verification problem: engineers still rely on manually written test cases. If scenarios describing all possible behaviors of a control could be generated *automatically* from the control definition, engineers could verify their intent by inspection, ensuring that the control captures exactly the intended policy.

Building such a DSL requires balancing two competing goals: the language must be **expressive** enough to encode complex compliance requirements (nested attributes, conditional logic, multiple resource types), yet **analyzable** enough for automated tools to reason precisely about control behavior. A highly expressive language may include constructs—unbounded loops, recursion, arbitrary computation—difficult or impossible to analyze automatically; an overly restricted language may be unable to express real-world requirements. Analyzability is the key enabler for automatic scenario generation, exhaustive testing, control comparison, and what-if analysis. *To our knowledge, no existing configuration control language provides automatic complete scenario and test case generation.*

`ControlsDSL`: an analyzable and expressive configuration control language. This paper presents `ControlsDSL`, a new declarative language for cloud configuration controls that is simultaneously expressive and analyzable. `ControlsDSL` is carefully designed so that its semantics can automatically be reduced to SMT formulas, enabling automated reasoning and test generation while remaining practical for real-world compliance requirements. `ControlsDSL`'s key innovation is automatic complete scenario generation: given a control, `ControlsDSL` produces human-readable configuration examples covering all execution paths, including edge cases that control authors might not consider. These scenarios serve as a verifiable specification—engineers can inspect them to confirm that the control captures their intended policy.

`ControlsDSL`'s JSON-based type system can naturally express compliance requirements based on resource attributes, tags, and conditional logic, leveraging AWS CloudFormation resource schemas, the lingua franca of AWS infrastructure-as-code [1, 2]. `ControlsDSL`'s design creates a safe foundation for authoring controls. `ControlsDSL` programs are purely declarative with no side effects, and the type system catches common mistakes such as referencing non-existent attributes or type mismatches. `ControlsDSL` automatically validates input resources against their declared types before evaluation, protecting controls against ill-typed inputs.

Using `ControlsDSL`'s formal semantics, we prove that our scenario generation is sound and complete. We evaluate `ControlsDSL` on practical compliance controls, demonstrating its expressiveness, the effectiveness of automatic scenario generation, and the practical value of SMT-based analysis for control development (Sec. 5).
In summary, this paper makes the following contributions:

(1) **Language Design**: `ControlsDSL`, a declarative DSL for cloud configuration controls with a JSON-based type system (Sec. 3).

(2) **Analyzability**: An SMT semantics for ControlsDSL that enables *(i)* a symbolic execution technique generating human-readable configuration scenarios that are sound and complete, *(ii)* automated test generation, infeasibility detection, and semantic control comparison (Sec. 4). All our analyses are accompanied by soundness and completeness theorems.

(3) **Evaluation**: An evaluation on 397 production controls at AWS, showing that ControlsDSL controls are 3.3× more concise than Python/Java equivalents, scenario generation completes in seconds, and SMT-based analysis achieves complete path coverage automatically (Sec. 5).

We discuss related work in Sec. 6 and conclude in Sec. 7.

2 Overview (ControlsDSL by Example)

This section introduces ControlsDSL through the example shown in Fig. 2. This example shows a ControlsDSL program that demonstrates key language constructs and, crucially, the configuration scenarios that ControlsDSL generates automatically. We use this example to build intuition before the formal treatment in Secs. 3 and 4.

Programs. A ControlsDSL *program* operates on a JSON object representing a cloud resource configuration. The program makes assertions about the values within this object. If any assertion fails, the resource is deemed *non-compliant*. Only when all assertions pass is the resource considered *compliant*.

A program starts with an INPUT statement that declares an input variable and its type.[2] In Fig. 2, the variable securityGroup has type AWS::EC2::SecurityGroup. A security group enforces the traffic that is allowed to reach and leave the resources associated with it [24]. An example of common control is to ensure that inbound rules do not permit resource access from any IP address, which occurs when CIDR IP is 0.0.0.0/0 or CIDR IPv6 is ::/0. Fig. 2 shows how to write this control in ControlsDSL.

The program requires iteration over the array of inbound rules, which ControlsDSL supports through the universal quantification construct EVERY. A key design constraint of ControlsDSL is that iteration bodies must be predicate calls. Without this restriction, iteration could contain arbitrarily complex logic, and the generated scenarios would inline the full logic of each loop body in every scenario exercising the assertion on the array—producing unwieldy, unreadable output. By requiring iteration bodies to be predicate calls, top-level scenarios reference only predicate names, while each predicate's internal logic is described separately in sub-scenarios.

The PREDICATE construct consists of a name, an argument with its type, and a body. The argument type is usually derived from the type alias of a resource type introduced by the USE construct, where [*] refers to the element type of an array. A predicate body can contain further iteration, referencing additional predicates.

Scenarios. There are only two main scenarios for this program: either every inbound rule satisfies SafeInboundRule (compliant), or some inbound rule does not (non-compliant). But what does it mean for an inbound rule to satisfy or not satisfy SafeInboundRule? To answer this, each predicate has its own independently generated *sub-scenarios*. For this program, SafeInboundRule has three sub-scenarios, SafeIpv4Range has two, and SafeIpv6Range has two. The first sub-scenario of SafeIpv4Range, for example, states that a range does not satisfy SafeIpv4Range if its CIDR IP is equal to 0.0.0.0/0.

This hierarchical structure keeps scenarios readable even for complex programs: an engineer can first review the main scenarios and then drill down into the sub-scenarios for each predicate. This approach addresses the scenario maintenance burden described in Sec. 1: instead of manually enumerating test cases, engineers inspect auto-generated scenarios at the appropriate level of abstraction.

SMT-based Features. Beyond scenario generation, ControlsDSL's carefully limited expressivity and its formal semantics enable encoding programs as SMT formulas (Sec. 4.2), which unlocks three additional capabilities. *Infeasible scenario detection* identifies and removes logically impossible scenarios by checking satisfiability of their path conditions. For instance, scenario generation may conservatively explore a path requiring range.cidrIp = "0.0.0.0/0" and range.cidrIp ≠ "0.0.0.0/0" simultaneously; SMT detects that this is unsatisfiable and removes the scenario. *Automatic test generation* uses SMT models to produce concrete JSON inputs that exercise each execution path, yielding complete test suites without manual effort. For example, given the scenario where range.cidrIp = "0.0.0.0/0", SMT produces a concrete JSON input such as {"cidrIp": "0.0.0.0/0"} that triggers the corresponding execution path. *Control comparison* verifies that two control versions (e.g., before/after refactoring) are semantically equivalent, or produces a counterexample input they disagree on.

3 The ControlsDSL Language

This section presents the formal syntax and types system (Sec. 3.1), and semantics (Sec. 3.2) of ControlsDSL. We then discuss key design decisions (Sec. 3.3) and state the language's core properties (Sec. 3.4). We use $\overline{\alpha}$ to denote that α can appear zero or more times.

3.1 Syntax and Types

The simplified syntax of ControlsDSL programs, types, and environments is defined in Fig. 3. This simplified syntax differs from the more comprehensive syntax used in Sec. 2, by omitting for simplicity some features supported by the complete language.

ControlsDSL uses JavaScript Object Notation (JSON) as the basis for its value representation (Booleans, strings, numbers, arrays, and objects).[3] Adopting JSON as the underlying data structure ensures compatibility and simplifies integration with cloud resources state descriptions. On top of JSON's primitive types, ControlsDSL also allows *enum types* of the form $\langle s_1, \ldots, s_j \rangle$, denoting a type consisting of j distinct string values.[4]

Every ControlsDSL program defines which subset of values from a specific resource type leads to compliant results. Resource types are themselves JSON types associated with a unique name identifier. ControlsDSL uses these identifiers as *named types* to

[2] In the actual implementation, we also support the ability to query input data from different *input sources*.

[3] This choice aligns with cloud APIs such as AWS, which return data in JSON format.

[4] Enum types are not used in the semantics of the language. Instead, they serve three purposes: (1) enhancing auto-completion in the IDE, (2) generating warnings about impossible matches during development, and (3) turning negative information (x is not one of [A, B, C]) into positive information (x is one of [D, E]) when scenarios are rendered into English text.

```
USE AWS::EC2::SecurityGroup AS SecurityGroup
INPUT securityGroup: AWS::EC2::SecurityGroup

PREDICATE SafeIpv4Range(
  range: SecurityGroup.config.ipPermissions[*].ipv4Ranges[*])
{ range.cidrIp != "0.0.0.0/0" }

PREDICATE SafeIpv6Range(
  range: SecurityGroup.config.ipPermissions[*].ipv6Ranges[*])
{ range.cidrIpv6 != "::/0" }

PREDICATE SafeInboundRule(
  inboundRule: SecurityGroup.config.ipPermissions[*]
) {
  EVERY range IN inboundRule.ipv4Ranges IS SafeIpv4Range(range) AND
  EVERY range IN inboundRule.ipv6Ranges IS SafeIpv6Range(range)
}

ASSERT EVERY inboundRule IN securityGroup.config.ipPermissions IS
  SafeInboundRule(inboundRule)
```

Main scenarios

✓ every inboundRule in `securityGroup.config.ipPermissions` is SafeInboundRule → COMPLIANT
✗ some inboundRule is not SafeInboundRule → NON_COMPLIANT

SafeInboundRule(*inboundRule*)

✓ every ipv4 range is SafeIpv4Range ∧ every ipv6 range is SafeIpv6Range → *satisfied*
✗ every ipv4 range is SafeIpv4Range ∧ some ipv6 range is not SafeIpv6Range → *not satisfied*
✗ some ipv4 range is not SafeIpv4Range → *not satisfied*

SafeIpv4Range(*range*)

✗ `range.cidrIp = "0.0.0.0/0"` → *not satisfied*
✓ `range.cidrIp ≠ "0.0.0.0/0"` → *satisfied*

SafeIpv6Range(*range*)

✗ `range.cidrIpv6 = "::/0"` → *not satisfied*
✓ `range.cidrIpv6 ≠ "::/0"` → *satisfied*

Figure 2: `ControlsDSL` rule ensuring a security group does not allow access from any IP address (left); hierarchical scenarios generated for each predicate (right).

Programs

$prog$	::= $\overline{pred}$ INPUT(x : τ){s}	*program*
$pred$	::= PRED a(x:τ){s}	*predicate*
s	::= CONST x:τ = e IN s	*statement*
	| ASSERT p ELSE r; s | ε	
r	::= true | false | ERROR	*result*
p	::= atom(b) | $p \wedge p$ | $p \vee p$	*proposition*
b	::= true | false | x | $e{=}e$ | $e{<}e$ | $\cdots$	*Boolean expression*
	| x.$f_1.\cdots.f_n$ EXISTS | a(e)	
	| x.$f_1.\cdots.f_n$ NOT EXISTS	
	| EVERY x $\in$ e. a(x)	
	| SOME x $\in$ e. a(x)	
e	::= n | σ | x | $e.f$ | [$\overline{e}$] | $F(\overline{e})$ | E.σ	*expression*

Types

τ	::= τ_{named} | τ_{prim} | τ_{object} | τ_{array}	*type*
τ_{named}	::= t	*named type*
τ_{prim}	::= bool | number | string | E⟨$\overline{\sigma}$⟩	*primitive type*
τ_{object}	::= {$\overline{f:\tau}$}	*object type*
τ_{array}	::= τ[ ]	*array type*

Figure 3: Syntax of `ControlsDSL`. We use x for variables, a for predicate names, t for type names, F for library function names, n for JSON numbers [26], σ for strings, f for field names, and E.σ for enumerative constants. The environments Γ_t, Γ_{fun}, Γ_{pred}, and Γ_{var} map type names, function names, predicate names, and variables to their types, respectively.

represent resource types such as `DynamoDB::Table` or `S3::Bucket`. We assume a type environment Γ_t, which maps every named type to a structural type and the signature of every library function F in the language to its type. For example, Γ_t(`DynamoDB::Table`) = {`Arn : str, name : str, ...`}. The type environment is an *input* to `ControlsDSL` rather than being hardcoded in the language or its interpreter: named types are derived automatically from the resource schemas maintained by the respective service owners (e.g., Config Recorder, CloudFormation, and API schemas), so new services and schema updates are onboarded without manual changes to the language.Named types can contain (mutually) recursive definitions, which allow `ControlsDSL` to support recursive JSON schemas. For example, one could define the following type: `Dir = {name : str, files : str[], dirs : Dir[ ]}`. `ControlsDSL` requires recursive type definitions to be well-founded and always admit a base case—e.g., the base case of the inductive definition of `Dir` is an empty array of `dirs`.

A `ControlsDSL` program consists of an input variable, zero or more predicate definitions, and a sequence of statements describing conditions under which values of the input variable result in compliance. A statement is either a variable binding (`CONST`) or an assertion (`ASSERT`). An input is *compliant* if it passes all assertions; otherwise the program returns a result $r \in$ {`false, ERROR`} when an assertion with the corresponding result r fails. To avoid notation clutter, the formalization here assumes all predicates take exactly one parameter as input—the quantified element of the collection on which the predicate is evaluated. It is straightforward to modify the formalization to allow more than one variable in the predicate signatures, and indeed the current version of the language supports additional parameters.

In `ControlsDSL`, assertions contain a *proposition* that can only be a positive Boolean combination of atomic Boolean operations (atom(b)). All branching logic that is relevant for scenario generation is confined to the propositional level. This separation is motivated by the desire to produce clean scenarios, as we detail in Sec. 4.1.

Fig. 3 contains a few representative Boolean operations. The ones worth noting are the following. First, the expression x.$f_1.\cdots.f_n$ `EXISTS` (resp. x.$f_1.\cdots.f_n$ `NOT EXISTS`) checks whether a specific sequence of field accesses to a variable exists (resp. does not exist). The quantifier expression `EVERY x ∈ e. a(x)` (resp. `SOME x ∈ e. a(x)`) checks whether all elements (resp. some element) of the array defined by an expression e satisfy the predicate a. Note that the predicate within a quantifier expression takes as input the quantified variable.

`ControlsDSL` does not allow negation of propositions and instead provide negated versions of Boolean operators where needed (e.g., `NOT IN` rather than negating the entire expression `e1 IN e2`). This design choice biases toward simplicity and safe defaults: absent evidence (such as a missing field), expressions evaluate to `false`.

Program typing

$$\frac{\Gamma_p = typesOf(\overline{pred}) \quad \forall pred_i \in \overline{pred}.(\Gamma_t, \Gamma_p \vdash_T pred_i : \tau_i \to \texttt{bool}) \quad \Gamma_t, \Gamma_p, \texttt{x} : \tau \vdash_T s \text{ ok}}{\Gamma_t \vdash_T \overline{pred}\ \texttt{INPUT(x} : \tau\texttt{)}\{s\} \text{ ok}}\ \text{t-prog}$$

Statement typing

$$\frac{\Gamma \vdash_T e : \tau \quad \Gamma, \texttt{x} : \tau \vdash_T s \text{ ok}}{\Gamma \vdash_T \texttt{CONST x} : \tau = e\ \texttt{IN}\ s \text{ ok}}\ \text{t-const} \qquad \frac{\Gamma \vdash_T p \text{ ok} \quad \Gamma \vdash_T s \text{ ok}}{\Gamma \vdash_T \texttt{ASSERT}\ p\ \texttt{ELSE}\ r; s \text{ ok}}\ \text{t-assert} \qquad \frac{}{\Gamma \vdash_T \varepsilon \text{ ok}}\ \text{t-emp-s}$$

Proposition typing

$$\frac{\Gamma \vdash_T b : \texttt{bool}}{\Gamma \vdash_T \texttt{atom}(b) \text{ ok}}\ \text{t-atom} \qquad \frac{\Gamma \vdash_T p_1 \text{ ok} \quad \Gamma \vdash_T p_2 \text{ ok}}{\Gamma \vdash_T p_1 \wedge p_2 \text{ ok}}\ \text{t-and} \qquad \frac{\Gamma \vdash_T p_1 \text{ ok} \quad \Gamma \vdash_T p_2 \text{ ok}}{\Gamma \vdash_T p_1 \vee p_2 \text{ ok}}\ \text{t-or}$$

Boolean Expression typing

$$\frac{\Gamma \vdash_T e_1 : \tau_{prim} \quad \Gamma \vdash_T e_2 : \tau_{prim}}{\Gamma \vdash_T e_1 = e_2 : \texttt{bool}}\ \text{t-prim-eq} \qquad \frac{\Gamma \vdash_T e_1 : \tau[\,]}{\Gamma \vdash_T e_1 = [\,] : \texttt{bool}}\ \text{t-empt-arr-eq}$$

$$\frac{\Gamma \vdash_T \texttt{a} : \tau \to \texttt{bool} \quad \Gamma \vdash_T e : \tau}{\Gamma \vdash_T \texttt{a(}e\texttt{)} : \texttt{bool}}\ \text{t-pred-app} \qquad \frac{\Gamma \vdash_T e : \tau[\,] \quad \Gamma \vdash_T \texttt{a} : \tau \to \texttt{bool}}{\Gamma \vdash_T \texttt{EVERY x} \in e.\ \texttt{a(x)} : \texttt{bool}}\ \text{t-all} \qquad \frac{\Gamma \vdash_T e : \tau[\,] \quad \Gamma \vdash_T \texttt{a} : \tau \to \texttt{bool}}{\Gamma \vdash_T \texttt{SOME x} \in e.\ \texttt{a(x)} : \texttt{bool}}\ \text{t-some}$$

Figure 4: Selected ControlsDSL typing rules. We use Γ to denote an arbitrary context in the typing judgments: $\Gamma \vdash_T (prog \mid s \mid p)$ ok ensures a program, statement, or proposition is well-typed under context Γ, $\Gamma \vdash_T pred : \tau \to$ bool ensures $pred$ is a predicate over variables of type τ, $\Gamma \vdash_T b :$ bool ensures Boolean expressions are well-typed, $\Gamma \vdash_T e : \tau$ ensures an expression e has type τ. Typing the main program requires knowing the types of predicates so that these can be called with the right parameters. The function $typesOf$ extracts a context consisting of the type signatures of a sequence of predicate definitions and is defined inductively as follows: $typesOf(\varepsilon) = \varepsilon$, $typesOf(pred\ \overline{pred}) = typesOf(pred), typesOf(\overline{pred})$, and $typesOf($PRED a $(\texttt{x} : \tau)\ \{p\}) = \texttt{a} : \tau \to$ bool.

ControlsDSL's type system (Fig. 4) is fairly standard; it enforces that JSON types are used correctly and that variables are in scope. All required definition are described in the caption of Fig. 4. The only point worth noting is that ControlsDSL limits equality checks to primitive types (τ_{prim}) and between an array-typed expression (τ_{array}) and the empty array (highlighted in Fig. 4). Object equality is deliberately disallowed because cloud resource schemas can evolve over time, rendering object equality fragile. Full programs are typed under a context Γ_t that contains all enumerative types and named type definitions—e.g., existing known resource types. This context is extended with variable definitions throughout the type system. A program is *well-typed* when it satisfies the typing judgment of Fig. 4 under Γ_t; the theorems in Secs. 3.4 and 4.1 apply to well-typed programs.

3.2 Semantics

The semantics of ControlsDSL is also fairly standard. Evaluation is expressed through judgments of the form $\sigma \vdash s \Downarrow r$, read as: under environment σ, statement s *evaluates to* result r (and analogously for propositions and expressions). A program applied to an input value v is written as follows: $\overline{pred}$ INPUT(x : τ){s}(v). The evaluation results in a result r, which can be true if the input v is compliant, false if the input is not compliant, and ERROR if an assertion in the program triggers an evaluation error. An assertion may "not pass" because its proposition either evaluates to false, or some field is missing during the evaluation.

For the security group example from Fig. 2, when securityGroup.config.ipPermissions is absent, the field access evaluates to MISS(securityGroup.config.ipPermissions), which propagates to false at the proposition level, causing the assertion to fail with a non-compliant result.

Evaluation rules for statements and propositions have a *short-circuit* semantics. Specifically, if an assertion is false, the evaluation stops, ignoring the remaining statements. Similarly, if the first disjunct of an OR is true, the second disjunct is not evaluated. The evaluation rules that produce MISS($e.f$) illustrate how ControlsDSL handles missing fields. In ControlsDSL, all record fields are optional. Accessing a field not defined in the record's type is a type error. However, accessing a field defined in the type but absent at runtime evaluates to false for the surrounding expression. This design biases evaluation away from compliance when evidence is lacking. Rather than requiring explicit field existence checks—which would lead to verbose rules—ControlsDSL implements this default behavior, in which evaluation can result in the value false due to missing fields. In Sec. 4.1, we will show how scenario generation exposes these implicit field-missing cases and allows rule authors to verify their intended behavior.

3.3 Design Decisions and Trade-offs

We summarize the most important of ControlsDSL's design decisions that involve trade-offs between expressiveness and analyzability. ControlsDSL requires that the body of a quantifier (EVERY, SOME) be a predicate call. This restriction enables readable hierarchical scenarios: because predicates are named, the scenarios for quantified expressions can reference predicate names rather than inlining complex conditions. ControlsDSL treats all record fields as optional, matching the real-world variability of JSON data. This

Values and environments

$v ::= \texttt{true} \mid \texttt{false} \mid n \mid \sigma \mid \{\overline{f : v}\} \mid [\overline{v}] \mid \texttt{E}.\sigma$ (value)

$\hat{v} ::= v \mid \texttt{MISS}(e.f)$ (extended value) $\quad r ::= \texttt{true} \mid \texttt{false} \mid \texttt{ERROR}$ (result) $\quad \mathcal{V} ::= \overline{\texttt{x} : v}; \overline{pred}$ (environment)

Program and statement evaluation

$$\frac{\overline{pred}; \texttt{x} : v \vdash s \Downarrow r}{\vdash \overline{pred}\ \texttt{INPUT}(\texttt{x} : \tau)\{s\}(v) \Downarrow r}\ \text{e-prog} \qquad \frac{\mathcal{V} \vdash e \Downarrow v \quad \mathcal{V}, \texttt{x} : v \vdash s \Downarrow r}{\mathcal{V} \vdash \texttt{CONST x} : \tau = e\ \texttt{IN}\ s \Downarrow r}\ \text{e-const} \qquad \frac{}{\varepsilon \Downarrow \texttt{true}}\ \text{e-empt-list}$$

$$\frac{\mathcal{V} \vdash p \Downarrow \texttt{true} \quad \mathcal{V} \vdash s \Downarrow r}{\mathcal{V} \vdash \texttt{ASSERT}\ p\ \texttt{ELSE}\ r; s \Downarrow r}\ \text{e-assert-true} \qquad \frac{\mathcal{V} \vdash p \Downarrow \texttt{false}}{\mathcal{V} \vdash \texttt{ASSERT}\ p\ \texttt{ELSE}\ r; s \Downarrow r}\ \text{e-assert-false}$$

Proposition evaluation

$$\frac{\mathcal{V} \vdash b \Downarrow \texttt{true}}{\mathcal{V} \vdash \texttt{atom}(b) \Downarrow \texttt{true}}\ \text{e-atom-true} \qquad \frac{\mathcal{V} \vdash b \Downarrow \texttt{false}}{\mathcal{V} \vdash \texttt{atom}(b) \Downarrow \texttt{false}}\ \text{e-atom-false} \qquad \frac{\mathcal{V} \vdash b \Downarrow \texttt{MISS}(e'.f)}{\mathcal{V} \vdash \texttt{atom}(b) \Downarrow \texttt{false}}\ \text{e-atom-missing}$$

$$\frac{\mathcal{V} \vdash p_1 \Downarrow \texttt{true} \quad \mathcal{V} \vdash p_2 \Downarrow v}{\mathcal{V} \vdash p_1 \wedge p_2 \Downarrow v}\ \text{e-and-true} \qquad \frac{\mathcal{V} \vdash p_1 \Downarrow \texttt{false}}{\mathcal{V} \vdash p_1 \wedge p_2 \Downarrow \texttt{false}}\ \text{e-and-false} \qquad \frac{\mathcal{V} \vdash p_1 \Downarrow \texttt{true}}{\mathcal{V} \vdash p_1 \vee p_2 \Downarrow \texttt{true}}\ \text{e-or-true}$$

Expression evaluation (selected rules)

$$\frac{\mathcal{V} \vdash e \Downarrow \{f_1 : v_1, \ldots, f_n : v_n\}}{\mathcal{V} \vdash e.f_i \Downarrow v_i}\ \text{e-field-access} \qquad \frac{\mathcal{V} \vdash e \Downarrow \{f_1 : v_1, \ldots, f_n : v_n\} \quad f \notin \{f_1, \ldots, f_n\}}{\mathcal{V} \vdash e.f \Downarrow \texttt{MISS}(e.f)}\ \text{e-field-missing}$$

$$\frac{\mathcal{V} \vdash e \Downarrow [v_1, \ldots, v_n] \quad \mathcal{V}, \texttt{x} : v_1 \vdash \texttt{a(x)} \Downarrow v_1' \quad \cdots \quad \mathcal{V}, \texttt{x} : v_n \vdash \texttt{a(x)} \Downarrow v_n'}{\mathcal{V} \vdash \texttt{EVERY x} \in e.\ \texttt{a(x)} \Downarrow \bigwedge_i v_i'}\ \text{e-all}$$

Figure 5: Selected evaluation rules for `ControlsDSL`. Values include JSON primitives, objects, and arrays; extended values add `MISSING`$(e.f)$ for absent fields. Our semantics uses the following evaluation judgments: $\mathcal{V} \vdash s \Downarrow r$ evaluates a statement in environment $\mathcal{V}$ to result r, $\mathcal{V} \vdash p \Downarrow v$ evaluates a proposition in environment $\mathcal{V}$ to Boolean value v, $\mathcal{V} \vdash (b \mid e) \Downarrow \hat{v}$ evaluates an expression (whether Boolean or not) in environment $\mathcal{V}$ to an extended value $\hat{v}$—i.e., either a value or a special value `MISSING`$(e.f)$ denoting a missing field. In all cases, $\mathcal{V}$ represents the program environment containing variable bindings, and the judgment $\Downarrow$ denotes evaluation to a final result.

design ensures that rules are robust against incomplete or evolving resource configurations. A predicate cannot call itself, directly or indirectly. This restriction ensures termination and decidability—both crucial for exhaustive scenario generation and SMT-based analysis. `ControlsDSL` disallows object equality because cloud resource schemas evolve over time: fields can be added or renamed. Array equality is allowed only with the empty array. This prevents brittle rules that could break as schemas evolve.

3.4 Deterministic Evaluation and Type Safety

We now state the core properties of `ControlsDSL`'s semantics. `ControlsDSL` satisfies standard type safety properties: well-typed programs always evaluate to a result (progress) and types are preserved during evaluation (preservation). More critically for compliance controls, `ControlsDSL`'s evaluation is deterministic.

Theorem 3.1 (Deterministic Evaluation). *Given an environment σ,* (i) *for any statement s and results r_1, r_2, if $\sigma \vdash s \Downarrow r_1$ and $\sigma \vdash s \Downarrow r_2$ then $r_1 = r_2$;* (ii) *for any expression e and extended values $\hat{v}_1, \hat{v}_2$, if $\sigma \vdash e \Downarrow \hat{v}_1$ and $\sigma \vdash e \Downarrow \hat{v}_2$ then $\hat{v}_1 = \hat{v}_2$.*

4 `ControlsDSL` Analysis

This section presents `ControlsDSL`'s two core analysis techniques. Sec. 4.1 describes *scenario generation*, which systematically explores all execution paths to produce complete, human-readable configuration scenarios. Because scenario generation conservatively explores all branches—including those whose conditions may be mutually contradictory—it can produce *infeasible* scenarios that no input can trigger. We address this first with a lightweight value-set propagation pass and then with precise SMT-based detection (Sec. 4.2.2). Sec. 4.2 then describes how `ControlsDSL` encodes programs and scenarios into SMT formulas, enabling infeasible scenario detection, automatic test generation, and semantic control comparison.

4.1 Scenario Generation

A scenario describes a potential execution path for a `ControlsDSL` program along with the expected result. Each scenario is a *path condition* together with a corresponding result.[5] A path condition is a list of conditions. Each condition says that a particular expression is true or false, that a particular field does or does not exist, or that a variable is bound to an expression (e.g., through `CONST` bindings).

When generating scenarios involving predicates, `ControlsDSL` adopts a modular approach where predicates are treated as symbolic values and their scenarios are generated separately.

[5] As discussed in Sec. 3.1, in this formalization, a scenario can only be mapped to the results `true`, `false`, and `ERROR`. In practice, every instance of `false` appearing in a `ControlsDSL` program has a corresponding annotation to denote different variants of "failures", including custom user messages to explain the reason for non-compliance.

Scenario generation thus produces two key outputs: a *predicate scenario mapping* (*PtoS*) that associates each predicate with its corresponding scenarios, and *main scenarios* (*SC*) in which predicates appear as free variables. We write $\vdash$ *program* $\Rightarrow$ *PtoS*; *SC* for the judgment stating that scenario generation applied to a program produces this pair of outputs, separated by a semicolon.

In the security group example from Fig. 2, for the main program, one scenario has a path condition stating that every inbound rule satisfies `SafeInboundRule`, leading to `TRUE`; the other states that some inbound rule does not, leading to `FALSE`. The predicate `SafeIpv4Range` generates two sub-scenarios: one where `range.cidrIp = "0.0.0.0/0"` ($\rightsquigarrow$ `FALSE`) and one where `range.cidrIp` $\neq$ `"0.0.0.0/0"` ($\rightsquigarrow$ `TRUE`).

Scenario generation (Fig. 6) systematically builds all possible execution paths through a `ControlsDSL` program. Intuitively, the rules symbolically execute a program to accumulate conditions under which the program may cause an assertion to fail and return `false` or `ERROR`, and the conditions under which all assertions pass and the scenario therefore maps to `true`. Scenario generation mimics the short-circuiting used in the evaluation semantics (Fig. 5), where Boolean operators are evaluated as early as possible.

The algorithm in Fig. 6 may generate logically infeasible scenarios—scenarios whose conditions cannot be satisfied by any well-typed input. This over-approximation can produce noise that rule authors must sift through. We address infeasibility at two levels. First, a lightweight *value-set propagation* pass tracks possible values during scenario construction and detects simple contradictions—for example, a scenario requiring a field to both exist and not exist. Second, for scenarios that survive value-set propagation, we can use SMT solving (Sec. 4.2) for precise detection: encoding the path conditions as an SMT formula and checking satisfiability. If the formula is unsatisfiable, the scenario is infeasible and can be removed.

The number of scenarios can be exponential in the depth of propositions, since each Boolean connective can double the number of paths. In practice, however, most rules produce a manageable number of scenarios for three reasons. First, scenarios terminate early when they reach any result other than compliant. Second, the language biases away from compliant results (for example, returning false when a field does not exist). Third, compliance rules tend to be conjunctive in nature—combining conditions conjunctively rather than disjunctively—naturally limiting the number of successful paths. Predicates further help by decomposing scenarios hierarchically: a quantified expression produces only two main scenarios (all/some), with the predicate's internal complexity captured in its own sub-scenarios.

We now proceed to state the main theorem characterizing why scenarios are correctly captured by `ControlsDSL`. We define judgments $\mathcal{V} \vdash \pi$ sat and $\mathcal{V} \vdash \Pi$ sat to express that a variable valuation $\mathcal{V}$ satisfies a condition π or a path condition Π, respectively. A path condition is satisfied when all its constituent conditions hold: a condition b holds when b evaluates to true; $\neg b$ holds when b evaluates to false; $\exists e.f$ holds when field f is present in e's value; $\neg\exists e.f$ holds when f is absent; and $\mathsf{x} = e$ holds when the variable is bound to e's value.

The following theorem combines deterministic evaluation (Theorem 3.1) with scenario generation soundness and completeness: every input is covered by exactly one scenario and always produces the result predicted by the associated scenario.

Theorem 4.1 (Fundamental Theorem of `ControlsDSL`). *For any well-typed* `ControlsDSL` *program* $program = \overline{pred}$ `INPUT`$(\mathsf{x} : \tau)\{s\}$ *and any input* $v : \tau$,

$$\vdash \overline{pred}\ \texttt{INPUT}(\mathsf{x} : \tau)\{s\} \Rightarrow PtoS; SC \implies$$
$$\exists sc = (\Pi \rightsquigarrow r_1) \in SC.\ (\mathsf{x} : v \vdash sc\ \text{sat})$$
$$\wedge\ (\forall r_2.\ \vdash \overline{pred}\ \texttt{INPUT}(\mathsf{x} : \tau)\{s\}(v) \Downarrow r_2 \implies r_1 = r_2)$$

The Fundamental Theorem captures the essence of `ControlsDSL`'s approach: instead of reasoning about evaluation semantics, engineers can reason about concrete scenarios—and our theorem guarantees consistency between these two views.

4.2 SMT-Based Analysis

`ControlsDSL`'s restricted semantics enables encoding programs as formulas in Satisfiability Modulo Theories (SMT), which we solve with Z3 [11]. The SMT encoding translates `ControlsDSL` types, expressions, and propositions into formulas over decidable SMT theories (Fig. 7), and unlocks three analysis capabilities previewed in Sec. 2: infeasible scenario detection, automatic test generation, and control comparison. We describe the encoding and each capability in turn.

Fig. 7 summarizes the SMT encoding of `ControlsDSL` programs, which preserves the semantics established in Sec. 3.2. We formalize this preservation as Theorem 4.2.

`ControlsDSL`'s JSON-based types map to SMT sorts as follows. Primitive types translate directly: `bool` to the SMT Boolean sort, `number` to the integer sort, and `string` (including enum types $\langle s_1, \ldots, s_k \rangle$) to the SMT string sort. The central challenge is encoding *object types with optional fields*. Since `ControlsDSL` treats all record fields as optional (Sec. 3.3), we cannot use a fixed SMT record type. Instead, we introduce an *uninterpreted sort* `Record` and, for each field f_i of type τ_i appearing in a record type, declare two uninterpreted functions: f_i`.EXISTS` : `Record` $\rightarrow$ `Bool`, which tracks whether the field is present, and $f_i.[\![\tau_i]\!]$: `Record` $\rightarrow [\![\tau_i]\!]$, which retrieves the field's value when present. This representation cleanly separates existence from value, mirroring `ControlsDSL`'s runtime semantics where accessing a missing field evaluates to `MISS` rather than producing an error. Array types $\tau[]$ are encoded using the SMT theory of arrays (`Array Int` $[\![\tau]\!]$), together with a function `arrayLength` that returns the number of elements. To bound the solver's search space and produce more compact tests, we initially constrain `arrayLength` to lie in $[0, N)$ for a configurable bound N; in case of `unsat` result, we relax the bound on N to soundly confirm the result is not an artifact of the bound. Named types are resolved through the type environment Γ_t before encoding. Recursive types are handled by bounded unrolling: the encoding unfolds recursive definitions up to a configurable depth, beyond which fields are treated as absent (matching `ControlsDSL`'s optional-field semantics).

The encoding of expressions follows the structure of the evaluation semantics (Sec. 3.2). Field access $e.f$ is encoded as a call to the value accessor $f.[\![\tau]\!]([\![e]\!])$, but any execution path that reaches this access is guarded by the corresponding existence flag

Scenarios and Path Conditions

$SC ::= 2^{sc}$ (scenarios) $sc ::= \Pi \rightsquigarrow r$ (scenario) $PtoS ::= PredName \to SC$ (predicate scenarios)
$\Pi ::= \varepsilon \mid \pi; \Pi$ (path condition) $\pi ::= b \mid \neg b \mid \exists e.f \mid \neg\exists e.f \mid \mathtt{x} = e$ (condition) $res ::= \mathtt{true} \mid \mathtt{false} \mid \mathtt{ERROR}$ (result)

Program generation

$$\frac{\vdash \overline{pred} \Rightarrow PtoS}{\vdash \overline{pred}\ \mathtt{INPUT(x}:\tau)\{s\} \Rightarrow PtoS; \{sc \mid \overline{pred}; x:\tau \vdash s \Rightarrow sc\}}\ \text{g-program}$$

Predicate list and predicate generation

$$\frac{}{\vdash \varepsilon \Rightarrow [\,]}\ \text{g-emp-p} \qquad \frac{\vdash pred \Rightarrow PtoS_1 \quad \vdash \overline{pred} \Rightarrow PtoS_2}{\vdash pred; \overline{pred} \Rightarrow PtoS_1 \cup PtoS_2}\ \text{g-cons-p} \qquad \frac{}{\vdash \mathtt{PRED\ a(x}:\tau)\{s\} \Rightarrow [\mathtt{a} \mapsto \{sc \mid \mathtt{x}:\tau \vdash s \Rightarrow sc\}]}\ \text{s-pred}$$

Statement generation

$$\frac{\mathtt{fields}(e) = [e_1, \ldots, e_n] \quad 0 \le i < n}{\Sigma \vdash \mathtt{CONST\ x}:\tau = e\ \mathtt{IN}\ s \Rightarrow (\exists e_1; \ldots; \exists e_i; \neg\exists e_{i+1}) \rightsquigarrow \mathtt{false}}\ \text{g-let-missing}$$

$$\frac{\mathtt{fields}(e) = [e_1, \ldots, e_n] \quad \Sigma, \mathtt{x}:\tau \vdash s \Rightarrow \Pi \rightsquigarrow r}{\Sigma \vdash \mathtt{CONST\ x}:\tau = e\ \mathtt{IN}\ s \Rightarrow (\exists e_1; \ldots; \exists e_n, \mathtt{x} = e, \Pi) \rightsquigarrow r}\ \text{g-let} \qquad \frac{}{\Sigma \vdash \varepsilon \Rightarrow \varepsilon \rightsquigarrow \mathtt{true}}\ \text{g-empty-list}$$

$$\frac{\Sigma \vdash p \Rightarrow \Pi_1 \rightsquigarrow \mathtt{true} \quad \Sigma \vdash s \Rightarrow \Pi_2 \rightsquigarrow r}{\Sigma \vdash \mathtt{ASSERT}\ p\ \mathtt{ELSE}\ r; s \Rightarrow (\Pi_1, \Pi_2) \rightsquigarrow r}\ \text{g-assert-true} \qquad \frac{\Sigma \vdash p \Rightarrow \Pi \rightsquigarrow \mathtt{false}}{\Sigma \vdash \mathtt{ASSERT}\ p\ \mathtt{ELSE}\ r; s \Rightarrow \Pi \rightsquigarrow r}\ \text{g-assert-false}$$

Proposition generation

$$\frac{\mathtt{fields}(b) = [e_1, \ldots, e_n] \quad 0 \le i < n}{\Sigma \vdash \mathtt{atom}(b) \Rightarrow (\exists e_1; \ldots; \exists e_i; \neg\exists e_{i+1}) \rightsquigarrow \mathtt{false}}\ \text{g-atom-missing}$$

$$\frac{\mathtt{fields}(b) = [e_1, \ldots, e_n]}{\Sigma \vdash \mathtt{atom}(b) \Rightarrow (\exists e_1; \ldots; \exists e_n; b) \rightsquigarrow \mathtt{true}}\ \text{g-atom-true}$$

$$\frac{\mathtt{fields}(b) = [e_1, \ldots, e_n]}{\Sigma \vdash \mathtt{atom}(b) \Rightarrow (\exists e_1; \ldots; \exists e_n; \neg b) \rightsquigarrow \mathtt{false}}\ \text{g-atom-false} \qquad \frac{\Sigma \vdash p_1 \Rightarrow \Pi_1 \rightsquigarrow \mathtt{true} \quad \Sigma \vdash p_2 \Rightarrow \Pi_2 \rightsquigarrow v}{\Sigma \vdash p_1 \wedge p_2 \Rightarrow (\Pi_1; \Pi_2) \rightsquigarrow v}\ \text{g-and-true}$$

$$\frac{\Sigma \vdash p_1 \Rightarrow \Pi_1 \rightsquigarrow \mathtt{false}}{\Sigma \vdash p_1 \wedge p_2 \Rightarrow \Pi_1 \rightsquigarrow \mathtt{false}}\ \text{g-and-false}$$

$$\frac{\Sigma \vdash p_1 \Rightarrow \Pi_1 \rightsquigarrow \mathtt{false} \quad \Sigma \vdash p_2 \Rightarrow \Pi_2 \rightsquigarrow v}{\Sigma \vdash p_1 \vee p_2 \Rightarrow (\Pi_1; \Pi_2) \rightsquigarrow v}\ \text{g-or-false} \qquad \frac{\Sigma \vdash p_1 \Rightarrow \Pi_1 \rightsquigarrow \mathtt{true}}{\Sigma \vdash p_1 \vee p_2 \Rightarrow \Pi_1 \rightsquigarrow \mathtt{true}}\ \text{g-or-true}$$

Figure 6: Scenario generation rules for `ControlsDSL` propositions: $\vdash prog \Rightarrow PtoS; SC$ holds when SC is the set of all possible scenarios for the program and $PtoS$ maps each predicate to its corresponding scenarios; $\Sigma \vdash s \Rightarrow cs \rightsquigarrow r$ holds when $cs \rightsquigarrow r$ is a valid scenario for statement s in context Σ: and $\Sigma \vdash p \Rightarrow cs \rightsquigarrow v$ holds when $cs \rightsquigarrow v$ is a valid scenario for proposition p in context Σ. Atomic propositions can lead to non-compliance (i.e., false) when some of their fields are missing (g-atom-missing) and when the underlying Boolean operation evaluates to false. Boolean operations generate "short-circuited" scenarios—e.g., if the left side of a disjunct evaluates to true, the corresponding evaluation is enough for that disjunct to generate a true scenario (g-or-true). The function `fields`(e) collects all field-access subexpressions (of the form $e.f$) from an expression e, traversing recursively through sub-expressions. The `fields` function plays a crucial role in generating scenarios for atomic propositions, as it identifies potential missing fields in the expression.

f.EXISTS($[\![e]\!]$). This guard mirrors scenario generation (Fig. 6), where the `fields` function identifies field accesses that may fail.

The IN operator (membership in a list) is encoded differently depending on whether the list is a constant or a variable. When the list is a constant $[c_1, \ldots, c_k]$, membership is expanded into a disjunction $(= [\![e]\!]\ c_1) \vee \cdots \vee (= [\![e]\!]\ c_k)$. When the list is a variable, membership becomes an existential quantifier over array indices.

Predicates require a translation step that bridges scenario generation and SMT encoding. For each predicate $\mathtt{a}(x : \tau)\{s\}$, we symbolically execute the body s using the scenario generation rules from Fig. 6 to collect all path conditions. The resulting scenarios are then partitioned by outcome and encoded as SMT functions $\mathtt{a}_{true}$ and $\mathtt{a}_{false}$ (see Fig. 7 for details). Since predicates cannot be recursive (Sec. 3.3), their call graph is a DAG, and we process predicates in *topological order*: a predicate is only encoded after all predicates it calls have already been translated. This ensures that every occurrence of $\mathtt{a}_{true}$ or $\mathtt{a}_{false}$ in a predicate body refers to an already-defined SMT function.

The quantifier expressions EVERY $x \in e$. $\mathtt{a}(x)$ and SOME $x \in e$. $\mathtt{a}(x)$ are encoded using the predicate functions defined above.

Type encoding $[\![\tau]\!]$

$[\![\texttt{bool}]\!] = \texttt{Bool}$ $[\![\texttt{number}]\!] = \texttt{Int}$
$[\![\texttt{string}]\!] = \texttt{String}$ $[\![\langle s_1, \ldots, s_k \rangle]\!] = \texttt{String}$
$[\![\{f_1 : \tau_1, \ldots\}]\!] = \texttt{Record}$ (uninterpreted sort)
$[\![\tau[\,]]\!] = (\texttt{Array Int } [\![\tau]\!])$

Field modeling for $\{f_1 : \tau_1, \ldots, f_n : \tau_n\}$

For each field $f_i : \tau_i$, declare:
f_i.EXISTS : Record → Bool *(existence)*
$f_i.[\![\tau_i]\!]$: Record → $[\![\tau_i]\!]$ *(value)*
For arrays: arrayLength : $[\![\tau_i]\!]$ → Int, bounded $[0, N)$

Expression encoding $[\![e]\!]$ (selected rules)

$[\![e.f]\!] = f.[\![\tau]\!]([\![e]\!])$ $[\![e.f \texttt{ EXISTS}]\!] = f.\texttt{EXISTS}([\![e]\!])$
$[\![e_1 = e_2]\!] = (= [\![e_1]\!]\ [\![e_2]\!])$ $[\![e_1 < e_2]\!] = (< [\![e_1]\!]\ [\![e_2]\!])$
$[\![e \texttt{ IN } [c_1, .., c_k]]\!] = \bigvee_j (= [\![e]\!]\ c_j)$
$[\![e_1 \texttt{ IN } e_2]\!] = \exists i.\ 0 \le i < \texttt{len}([\![e_2]\!]) \wedge [\![e_1]\!] = \texttt{select}([\![e_2]\!], i)$

Quantifier encoding

$[\![\texttt{EVERY } x \in e.\ \texttt{a}(x)]\!] = \forall i.\ (0 \le i < \texttt{len}([\![e]\!])) \Rightarrow \texttt{a}_{true}(\texttt{sel}([\![e]\!], i))$
$[\![\texttt{SOME } x \in e.\ \texttt{a}(x)]\!] = \exists i.\ 0 \le i < \texttt{len}([\![e]\!]) \wedge \texttt{a}_{true}(\texttt{sel}([\![e]\!], i))$

Predicate encoding

For predicate $\texttt{a}(x : \tau)\{s\}$, collect path conditions:
$\texttt{a}_{true}(x) \triangleq \bigvee_{\{\Pi \mid s \Rightarrow \Pi \rightsquigarrow \texttt{true}\}} [\![\Pi]\!]$
$\texttt{a}_{false}(x) \triangleq \bigvee_{\{\Pi \mid s \Rightarrow \Pi \rightsquigarrow \texttt{false}\}} [\![\Pi]\!]$
Processed in topological order of the call graph.

Figure 7: SMT encoding for ControlsDSL types, expressions, quantifiers, and predicates. Objects use an uninterpreted Record sort with per-field existence flags and value accessors. Arrays use bounded lengths. Predicates compile to SMT functions via scenario generation path conditions.

When the array bound N is small, to improve performance, quantifiers can be *unrolled* into a finite conjunction (for EVERY) or disjunction (for SOME) over all positions. For example, EVERY $x \in e.\ \texttt{a}(x)$ with bound N becomes $\bigwedge_{i=0}^{N-1} \left(i < \texttt{arrayLength}([\![e]\!]) \implies \texttt{a}_{true}(\texttt{select } [\![e]\!]\ i)\right)$. Bounded unrolling eliminates quantifier reasoning entirely, reducing the problem to quantifier-free theories, at the cost of a larger formula. In practice, we find that bounded unrolling with moderate bounds (e.g., $N < 10$) provides a good trade-off between formula size and solver performance (as discussed above, unsat results under a bound are confirmed by relaxing it).

4.2.1 Decidability. The language restrictions from Sec. 3.3 ensure that all SMT formulas produced by the encoding fall within decidable theory fragments. Specifically, no general recursion ensures finite unrolling; equality restricted to primitives avoids undecidable algebraic datatype fragments; ControlsDSL's string operations (equality, prefix/suffix, containment, regex matching) fall within the decidable fragment of SMT string theories [11]; and bounded array lengths allow quantifiers to be reduced to quantifier-free formulas. The encoding targets a combination of QF_LIA, the SMT-LIB string theory, bounded arrays, and uninterpreted functions, ensuring that all well-typed ControlsDSL programs yield decidable formulas.

Theorem 4.2 (SMT Encoding Soundness). *For any well-typed* ControlsDSL *program* $program = \overline{pred}\ \texttt{INPUT}(\texttt{x} : \tau)\{s\}$ *and any input* $v : \tau$,

$$\vdash \overline{pred}\ \texttt{INPUT}(\texttt{x} : \tau)\{s\} \Rightarrow PtoS; SC \implies \forall r.\ \left(\vdash \overline{pred}\ \texttt{INPUT}(\texttt{x} : \tau)\{s\}(v) \Downarrow r\right) \iff \left(\exists (\Pi \rightsquigarrow r) \in SC.\ \text{SAT}\left([\![\Pi]\!][\texttt{x} \mapsto [\![v]\!]]\right)\right)$$

The theorem states that the SMT encoding faithfully represents the program's input–output behavior: a program evaluates to result r on input v if and only if there exists a scenario in SC mapping to r whose encoded path condition is satisfiable when the input variable is bound to the encoding of v. Together with the Fundamental Theorem (Theorem 4.1), this establishes a chain from evaluation semantics through scenarios to SMT formulas, ensuring that all three representations agree on every well-typed input.

4.2.2 Infeasible Scenario Detection via SMT. Given a scenario $sc = (\Pi \rightsquigarrow r)$, we encode each condition in the path Π as an SMT assertion and check satisfiability of their conjunction. If the conjunction is unsatisfiable, no well-typed input can satisfy all conditions simultaneously, and the scenario is infeasible.

This SMT-based approach is strictly more precise than the lightweight value-set propagation described earlier. For example, value-set propagation cannot detect that a scenario requiring x.name STARTS_WITH "prod-" and x.name = "dev-instance" is infeasible, since that requires reasoning about the semantics of string prefix operations; SMT solving handles this via string theory.

In practice, infeasibility checking also serves as an *early pruning* mechanism during test generation (Sec. 4.2.3). When the tool explores a partial execution path, it checks whether the accumulated path conditions are still satisfiable. If an intermediate check returns unsatisfiable, the entire subtree of continuations is pruned, avoiding unnecessary exploration.

4.2.3 Automatic Test Generation. When an SMT query for a scenario is satisfiable, the solver produces a *model*—a concrete assignment of values to all symbolic variables. We extract this model and translate it back into a JSON object conforming to the input type. This JSON object serves as a test input that exercises the corresponding execution path.

Because scenarios are complete (Theorem 4.1), generating one test per feasible scenario yields a test suite with complete path coverage: every reachable execution path of the program is exercised by at least one test. This eliminates the need for manual test authoring and provides a formal coverage guarantee that is difficult to achieve with hand-written tests. The generated tests can be exported in standard formats and integrated into CI/CD pipelines.

4.2.4 Control Comparison. A common need during control authoring and refactoring is to verify that a modified control behaves identically to the original. Given two controls $ctrl_1$ and $ctrl_2$ over the same input type, we encode both as SMT formulas. Specifically, each control's scenarios are encoded as path conditions with their associated results, so that the formula captures the complete input–output behavior of the control. We then check $\exists v.\ ctrl_1(v) \neq ctrl_2(v)$. If this formula is *unsatisfiable*, the two controls are semantically equivalent—they produce the same result for every well-typed input.

Table 1: LOC: original (Python/Java) vs. `ControlsDSL` migrated.

		Median LOC		CD/Orig. Ratio	
Language	n	Orig.	CD	Med.	Mean
Java	17	49	14	0.29	0.37
Python	69	45	16	0.31	0.34
Overall	86	48	16	0.30	0.35

Table 2: Statistics for scenario generation across all 397 rules.

	Migrated (n = 86)			Non-Migrated (n = 311)		
Metric	Med	Mean	Max	Med	Mean	Max
CD LOC	16	18.2	64	15	13.0	52
Predicates	0	0.8	4	0	0.1	6
Main scenarios	6	14.9	179	5	4.6	12
Pred. scenarios	0	4.3	73	0	0.3	33
Gen. time (ms)	862	1,001	6,487	856	858	1,071

If the formula is *satisfiable*, the solver produces a counterexample: a concrete input on which the controls disagree.

These capabilities are particularly valuable for migration scenarios, where Python or Java controls are being rewritten in `ControlsDSL` and engineers need confidence that the translation preserves behavior. For these scenarios, we implemented a tailored symbolic execution engine that extracts SMT-encoded path conditions from the Python/Java controls as well. However, the symbolic execution of Python and Java controls is beyond the scope of this paper.

5 Evaluation

We evaluate `ControlsDSL` on a corpus of compliance controls deployed at Amazon Web Services. Our evaluation addresses the following three research questions:

RQ1 (Expressiveness): Can `ControlsDSL` express real-world compliance controls, and how does it compare to general-purpose implementations?

RQ2 (Scenario Generation): Is automatic scenario generation practical: are the outputs manageable and the analysis fast?

RQ3 (Test Generation): Can SMT-based analysis effectively generate test inputs for compliance controls?

5.1 Dataset and Deployment

Our evaluation dataset comprises 397 `ControlsDSL` controls (5,599 total LOC) deployed in production at AWS. Of these, 86 were migrated from existing Python (69) and Java (17) implementations, and 311 were written directly in `ControlsDSL` by control authors. These controls are evaluated over one billion times per day across all AWS commercial regions, checking the compliance of customer cloud resources against security best practices and configuration policies.

The migrated controls provide a controlled comparison: each has a pre-existing implementation in a general-purpose language whose behavior serves as ground truth. The natively-written controls demonstrate that `ControlsDSL` is practical for new control development beyond migration. For the migrated controls, we additionally collected SMT encoding metrics and solver performance data to evaluate the SMT-based analyses described in Sec. 4.2. In the figures, we use CD as short for `ControlsDSL`.

5.2 RQ1: Expressiveness

Table 1 summarizes the lines-of-code comparison. `ControlsDSL` controls are substantially more concise than their general-purpose counterparts: the median `ControlsDSL` control is 16 lines, compared to 48 lines for the original Python or Java implementation—a 3.3× reduction. The conciseness is consistent across source languages, with median ratios of 0.31 for Python and 0.29 for Java. This reduction reflects `ControlsDSL`'s design: built-in constructs for field access, existence checks, and assertion patterns eliminate boilerplate that general-purpose languages require for JSON traversal, error handling, and result construction.

Beyond migration, 311 additional controls have been written directly in `ControlsDSL`, bringing the total to 397 deployed controls covering a broad range of AWS resource types and policies.

To answer RQ1: `ControlsDSL` successfully expresses all production compliance controls in our study with 3.3× fewer lines of code than general-purpose implementations. In our experience, this conciseness eases authoring, review, and maintenance, although we have not run a controlled study to quantify these effects yet.

5.3 RQ2: Scenario Generation

Table 2 presents statistics on scenario generation across all 397 controls. The typical control produces a small number of scenarios: the median is 6 main scenarios for migrated controls and 5 for natively-written controls. The maximum for non-migrated controls is 12, suggesting that controls written directly in `ControlsDSL` tend to be simpler—though this is an observational comparison: migrated controls may have been more complex to begin with. The migrated controls are occasionally more complex, reaching 179 main scenarios for one control (`db-instance-backup-enabled`, a control with many interacting optional parameters).

Scenario generation is fast. Across all 397 controls, the median generation time is 857 ms, with the slowest control completing in 6.5 seconds. The time includes parsing, type checking, scenario generation, value-set propagation for infeasibility detection, and output formatting. These times are practical for interactive use during rule development.

Not all generated scenarios are feasible: some paths contain contradictory constraints that no input can satisfy simultaneously. Using the SMT-based infeasibility detection described in Sec. 4.2.2, we identified 142 infeasible scenarios out of 1,279 main scenarios across the 86 migrated rules (11.1%), plus 13 infeasible predicate scenarios out of 351. Nine controls had at least one infeasible main scenario; in the most extreme case (`elasticsearch-logs-to-cloudwatch`), 70/141 scenarios (50%) were infeasible. Without SMT-based pruning, control authors would need to manually review and dismiss these spurious scenarios, undermining the utility of the scenario output.

To answer RQ2: Scenario generation is practical for production use, producing manageable outputs in interactive time. SMT-based infeasibility detection is essential, eliminating spurious scenarios that would otherwise reduce the utility of generated outputs.

Table 3: SMT encoding and analysis metrics for the 86 migrated controls.

Metric	Min	Median	Mean	Max
SMT-LIB LOC	9	748	4,775	112,831
Declarations	1	56	175	2,311
Assertions	0	31	176	3,008
Quantifiers (35 rules)	0	0	61	1,016
Tests generated	5	30	116.2	2,406
SMT analysis time (ms)	1	20	24,497	1,465,361

5.4 RQ3: SMT-Based Test Generation

For each migrated control, the SMT encoding described in Sec. 4.2 is used to generate one test input per feasible scenario, yielding a test suite with complete path coverage by construction (Sec. 4.2.3). Table 3 summarizes encoding complexity and solver performance.

The median SMT-LIB encoding is 748 lines with 56 declarations and 31 assertions. The largest encoding exceeds 112K lines (for a rule with deeply nested types and many array-typed fields), but these are handled transparently by the tool. Of the 86 rules, 35 require quantified formulas (rules using `EVERY` or `SOME` over arrays).

The tool generates a median of 30 test inputs per control and up to 2,406 for the most complex control (by default, test generation expands paths also within predicate calls, hence the total number of tests may match the product of main scenarios and the scenarios within each called predicate). Because scenarios are complete (Theorem 4.1), the generated test suite covers every reachable path—a coverage guarantee difficult to achieve with hand-written tests.

The total SMT analysis time per control has a median of 20 ms. The distribution has a heavy tail: the 90th percentile is 5.7 s and the 95th percentile is 19.6 s, while the single slowest rule requires approximately 24 minutes. These outliers correspond to controls with large quantifier counts (up to 1,016) and deeply nested types, where the solver's search space grows substantially. For practical use, the tool applies early pruning—checking satisfiability of partial path conditions and pruning infeasible subtrees—which reduces the overall number of solver calls.

During migration, SMT-based control comparison (Sec. 4.2.4) surfaced behavioral discrepancies between the original Python/Java controls and their `ControlsDSL` iteratively developed versions, which were resolved during development, before the new controls's deployment. Examples of recurring discrepancies include *(i)* implicit general-purpose-language semantics—e.g., in Python an empty list, an empty string, or the integer 0 evaluate to false, conditions that `ControlsDSL` requires to be stated explicitly—and *(ii)* implicit schema assumptions—e.g., an original control assumed a list of resource identifiers contained no duplicates by construction; the solver, unconstrained by such tacit knowledge, exposed it as a counterexample to equivalence, prompting the authors to encode the assumption explicitly. Beyond validating the migrated controls, these checks make implicit assumptions visible and explicit, hardening controls in the process.

To answer RQ3: SMT-based test generation effectively automates test suite creation with complete path coverage by construction—a guarantee unattainable with manual testing. This eliminates test-writing effort while providing stronger correctness guarantees, improving both productivity and confidence in deployed controls.

5.5 Threats to Validity

The correctness of our analyses relies on the implementation faithfully following the formal rules; we mitigate this through extensive testing, validating all generated test inputs against the concrete evaluator. The rules in our dataset are also deployed and monitored in production, serving over one billion requests per day on average. Our evaluation focuses on AWS compliance controls; while the underlying principles may generalize to other domains (e.g., Kubernetes policies or other cloud providers), this remains to be validated. We measure expressiveness via LOC reduction and scenario quality via counts and infeasibility rates rather than a user study; LOC is an imperfect proxy—it partly reflects eliminated boilerplate rather than expressive power—and the comparison between migrated and natively-authored controls is observational, not controlled. We work closely with `ControlsDSL` program authors to improve the expressiveness and ergonomics of the language.

6 Related Work

Policy, configuration, and compliance languages. Cedar [4] is an AWS authorization policy language with formal foundations, but targets authorization decisions rather than resource compliance. Rego [23] (Open Policy Agent) is a Datalog-based policy language; CUE [9], Dhall [12], and Jsonnet [15] focus on configuration generation and validation. AWS Config Rules [3] allow defining compliance checks as Python Lambda functions—the approach `ControlsDSL` replaces. HashiCorp Sentinel [16] and Chef InSpec [21] provide policy-as-code with embedded DSLs. None of these tools generate complete scenarios, detect infeasible paths, or support formal equivalence checking. `ControlsDSL` combines a domain-specific compliance language with automatic scenario generation and SMT-based analysis, backed by formal guarantees (Sec. 3.4).

Formal methods for cloud security. Zelkova [6] uses SMT solving to analyze AWS IAM policies; Tiros [5] reasons about network reachability; Block Public Access [7] verifies S3 access configurations; and IAM-PolicyRefiner [10] synthesizes least-permissive IAM policies. These tools target specific security aspects (authorization, networking, access control), whereas `ControlsDSL` addresses the broader domain of resource configuration compliance. `ControlsDSL`'s scenario generation provides a novel mechanism for making compliance rules *inspectable*, which prior tools do not address.

Symbolic execution and verification DSLs. Symbolic execution [18] has a long history, with tools such as DART [13], KLEE [8], SPF [22], and SAGE [14] that typically face path explosion from loops and recursion. `ControlsDSL` avoids this by restricting the language (no general recursion, no unbounded loops), making scenario generation and SMT encoding *exhaustive* while keeping scenario counts manageable. Specification languages like Alloy [17] and TLA+ [19], and verification-oriented languages like Dafny [20] and F* [25], target expert users writing formal specifications. `ControlsDSL` targets compliance engineers who are not formal methods experts: by restricting the language sufficiently, specifications (scenarios) and test cases are *derived automatically* from programs, removing the need for users to write them manually.

7 Conclusion

We introduced `ControlsDSL`, a domain-specific language for cloud compliance controls whose key innovation is *automatic scenario generation* with formally proved *soundness* and *completeness*: for every well-typed input, exactly one scenario matches and correctly predicts the control's outcome (Theorem 4.1), enabling engineers to verify correctness by inspecting scenarios rather than reasoning about code. `ControlsDSL`'s restricted semantics enables SMT-based infeasibility detection, test generation with complete path coverage, and semantic equivalence checking between control versions.

Our evaluation on 397 controls deployed in production at AWS—serving over one billion daily compliance evaluations—shows that `ControlsDSL` controls are 3.3× more concise than their Python/Java counterparts, scenario generation completes in under one second for the vast majority of controls, and SMT-based test generation achieves complete path coverage automatically.

Future directions include mechanizing the metatheory in a proof assistant, extending `ControlsDSL` to support richer data types and other compliance domains including multi-cloud and Infrastructure-as-Code.

Data Availability Statement. `ControlsDSL`, its tools, and the controls authored in `ControlsDSL` are not currently open-source. The controls are managed by AWS and customers can enable them for their deployments via AWS Control's dashboard.

References

[1] Amazon Web Services. 2024. AWS CloudFormation Resource Type Reference. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html. Accessed: 2026-02-09.

[2] Amazon Web Services. 2024. AWS CloudFormation User Guide. https://docs.aws.amazon.com/cloudformation/. Accessed: 2026-02-09.

[3] Amazon Web Services. 2024. AWS Config Rules. https://docs.aws.amazon.com/config/latest/developerguide/evaluate-config_use-rules.html

[4] Amazon Web Services. 2024. Cedar. https://www.cedarpolicy.com

[5] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and Blake Whaley. 2019. Reachability Analysis for AWS-Based Networks. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, 231–241. doi:10.1007/978-3-030-25543-5_14

[6] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. 1–9. doi:10.23919/FMCAD.2018.8602994

[7] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Dan Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. 2020. Block public access: trust safety verification of access control policies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. 281–291. doi:10.1145/3368089.3409728

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 209–224.

[9] CUE Project. 2024. CUE – Configure, Unify, Execute. https://cuelang.org/

[10] Loris D'Antoni, Shuo Ding, Amit Goel, Mathangi Ramesh, Neha Rungta, and Chungha Sung. 2024. Automatically Reducing Privilege for Access Control Policies. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024). doi:10.1145/3689738

[11] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 4963)*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[12] Dhall Contributors. 2024. Dhall – A programmable configuration language. https://dhall-lang.org/

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 213–223. doi:10.1145/1065010.1065036

[14] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. In *Queue*, Vol. 10. 20–27. doi:10.1145/2090147.2094081

[15] Google. 2024. Jsonnet – A data templating language. https://jsonnet.org/

[16] HashiCorp. 2024. Sentinel – Policy as Code Framework. https://www.hashicorp.com/sentinel

[17] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis* (revised ed.). MIT Press.

[18] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394. doi:10.1145/360248.360252

[19] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

[20] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (LNCS, Vol. 6355)*. Springer, 348–370. doi:10.1007/978-3-642-17511-4_20

[21] Progress Software. 2024. Chef InSpec – Compliance as Code. https://www.chef.io/products/chef-inspec

[22] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 179—-180. doi:10.1145/1858996.1859035

[23] Rego. 2024. Rego. https://www.openpolicyagent.org/docs/latest/policy-language/

[24] Amazon Web Services. 2026. Control traffic to your AWS resources using security groups. https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html

[25] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 256–270. doi:10.1145/2837614.2837655

[26] W3. 2023. JSON Data Types. https://www.w3schools.com/js/js_json_datatypes.asp

Received 2026-04-29; accepted 2026-07-01