PROVISIONAL PATENT APPLICATION

FUSED SEMANTIC EXECUTION ENGINE FOR DETERMINISTIC STREAMING RULE EVALUATION

INVENTOR

Michael Allen Kuykendall
Lenexa, Kansas, United States
United States Citizen

Filing Date: _____

FIELD OF THE INVENTION

[0001] The present invention relates generally to data stream processing systems, and more particularly to a deterministic, single-pass execution engine that compiles multiple independent decision rules into a fused executable program evaluated during data ingestion, enabling bounded-memory, early-exit decisions without constructing intermediate representations.

BACKGROUND OF THE INVENTION

[0002] Modern computing systems frequently require the evaluation of multiple decision rules against streaming data. Applications include API gateway policy enforcement, real-time log alerting, security monitoring, financial transaction screening, event-driven architectures, network packet inspection, database query optimization, schema validation, content moderation, IoT sensor processing, and machine learning feature extraction. In such systems, data arrives continuously as a stream of structured records (e.g., JSON documents, log entries, network packets, database rows, sensor readings), and multiple independent rules must be evaluated against each record to produce decisions such as allow/deny, alert/suppress, route/drop, or classify/extract.

[0003] Conventional approaches to multi-rule evaluation suffer from fundamental architectural limitations that result in suboptimal performance characteristics as the number of rules increases or as data volumes grow.

Sequential Rule Evaluation

[0004] The most common approach evaluates rules sequentially, where each rule independently parses or traverses the input data to extract relevant values and evaluate predicates. This results in O(N×M) complexity, where N is the number of rules and M is the size of the input data. Each rule redundantly performs parsing and field extraction operations, even when multiple rules reference the same data paths. Examples include Open Policy Agent (OPA), AWS Cedar, Envoy proxy filter chains, and most web application firewalls (WAFs) using ModSecurity or similar rule languages.

Runtime Interpretation

[0005] Many policy and rule engines interpret rule definitions at runtime, incurring interpretation overhead on every evaluation. While some systems compile rules to intermediate representations such as WebAssembly (WASM), the compilation typically occurs per-rule rather than across rules, preserving the O(N) scaling with rule count during evaluation.

Intermediate Representation Construction

[0006] Traditional parsing approaches construct complete intermediate representations—such as Document Object Models (DOMs), Abstract Syntax Trees (ASTs), or in-memory object graphs—before rule evaluation can begin. This approach requires: (a) memory allocation proportional to document size; (b) complete document receipt before evaluation can start; (c)

inability to make early decisions based on partial data; and (d) multiple traversals when rules access different portions of the structure.

State-Heavy Matching Networks

[0007] The Rete algorithm and its derivatives (Rete-II, Phreak) used in business rule management systems (Drools, CLIPS) maintain complex networks of partial match state in working memory. While these approaches achieve some sharing of intermediate computations, they require pre-loading of facts into working memory before evaluation, maintain persistent state between evaluations, and exhibit memory consumption that grows with both rule count and data volume. These systems are designed for knowledge-base reasoning rather than streaming data processing.

Pattern-Only Fusion

[0008] Some systems achieve efficient multi-pattern matching through automaton fusion. Intel Hyperscan, for example, compiles multiple regular expressions into a fused deterministic finite automaton (DFA) that evaluates all patterns in a single pass. However, these systems are limited to pattern matching (regular expressions, literal strings) and cannot evaluate semantic predicates involving numeric comparisons, type checking, structural validation, or conditional logic based on extracted values.

The Unsolved Problem

[0009] What is needed, and what has not been achieved in the prior art, is a system that combines: (a) streaming evaluation during data ingestion without intermediate representation construction; (b) compile-time fusion of multiple independent semantic rules into a single execution program; (c) evaluation cost that is constant or near-constant with respect to rule count after compilation; (d) deterministic, fail-closed behavior on malformed or incomplete input; (e) early termination capability based on partial data when sufficient information exists to resolve all pending decisions; and (f) general-purpose applicability across diverse rule types including existence checks, numeric comparisons, type validation, and boolean combinations thereof.

[0010] The present invention addresses these limitations by providing a fused semantic execution engine that achieves the combination of properties described above.

SUMMARY OF THE INVENTION

[0011] The present invention provides a fused semantic execution engine comprising a compiler, an execution module, and a data element provider that together enable deterministic evaluation of multiple independent decision rules in a single streaming pass over input data, without constructing intermediate representations and with evaluation cost that is independent of the number of compiled rules.

[0012] In accordance with one aspect of the invention, a method for evaluating multiple decision rules against streaming structured data comprises: receiving a plurality of independent decision rules, each rule specifying one or more selectors identifying paths within structured data and one or more predicates to evaluate against values at those paths;

compiling the plurality of rules into a single fused executable program wherein selectors common to multiple rules are deduplicated and predicates are organized by selector; receiving structured data elements with associated path information during data ingestion without constructing an intermediate representation of the complete data; for each data element, dispatching execution to instructions associated with the current data path using a selector dispatch mechanism; evaluating predicates and updating rule resolution state in a rule state data structure; and terminating evaluation early when all rules have been resolved, before complete consumption of the input.

[0013] In accordance with another aspect of the invention, a system for fused semantic execution comprises: a compiler module configured to parse rule definitions, construct a selector data structure from all unique selectors across all rules, and emit a fused executable program with selector-indexed dispatch tables; a data element provider configured to incrementally provide structured data elements with associated path information during data ingestion; an execution module configured to receive data elements, perform dispatch to relevant instructions based on current path, evaluate instructions, and maintain rule resolution state; and an early exit mechanism that monitors rule resolution state and terminates evaluation when all rules are determined.

[0014] The invention achieves several technical advantages over the prior art. First, evaluation complexity is O(M) with respect to input size and independent of total rule count for rules sharing existing selectors, enabling systems with thousands of rules to evaluate at the same speed as systems with few rules. Second, streaming evaluation without intermediate representation construction enables bounded memory operation with memory requirements determined at compile time. Third, early exit capability enables decisions to be made before complete data receipt when partial data is sufficient. Fourth, deterministic fail-closed semantics ensure predictable behavior on malformed or incomplete input.

[0015] The fused executable program may be embodied as bytecode interpreted by a virtual machine, native machine code produced by just-in-time or ahead-of-time compilation, a decision graph or automaton structure, hardware description language for FPGA or ASIC implementation, or other executable representations. The data element provider may be embodied as a streaming tokenizer for text-based formats, a schema-driven deserializer for binary formats, a memory-mapped data accessor, a database cursor or row stream, or other mechanisms that provide structured data elements with path context.

BRIEF DESCRIPTION OF THE DRAWINGS

[0016] FIG. 1 is a block diagram illustrating the overall system architecture of the fused semantic execution engine in accordance with an embodiment of the present invention.

[0017] FIG. 2 is a flowchart illustrating the compilation process that transforms independent rules into a fused executable program.

[0018] FIG. 3 is a diagram illustrating the structure of the selector prefix tree (trie) used for efficient path-to-instruction dispatch.

[0019] FIG. 4 is a diagram illustrating a bytecode instruction format and representative instruction set in one embodiment.

[0020] FIG. 5 is a flowchart illustrating the streaming evaluation process including element dispatch and early exit determination.

[0021] FIG. 6 is a diagram illustrating the rule state data structure and its evolution during evaluation.

[0022] FIG. 7 is a diagram illustrating the integration between a data element provider and the execution module.

[0023] FIG. 8 is a comparison diagram illustrating the difference between conventional sequential evaluation and fused evaluation in accordance with the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0024] The following detailed description sets forth specific embodiments of the fused semantic execution engine. It will be apparent to those skilled in the art that the invention may be practiced in other embodiments. The detailed description and specific examples are provided for illustration only and are not intended to limit the scope of the invention.

System Overview

[0025] Referring now to FIG. 1, the fused semantic execution engine 100 comprises three primary components: a compiler 110, a data element provider 120, and an execution module 130. The system receives as input a set of rule definitions 102 and a stream of structured data 104, and produces as output a set of rule decisions 106.

[0026] The compiler 110 receives rule definitions 102 and produces a fused executable program 112. The compilation process occurs once, at system initialization or when rules change, and the resulting executable program is reused across all subsequent evaluations. The compiler performs selector deduplication, predicate optimization, and instruction emission as described in detail below.

[0027] The data element provider 120 receives the input data stream 104 and produces a sequence of data elements with path context 122. The provider operates incrementally, providing elements as input arrives without waiting for complete data receipt. Each data element includes the current structural path (e.g., depth, key name, array index), the element type (e.g., start of object, end of object, key, value), and for value elements, the value content. In a preferred embodiment, the data element provider is implemented as a streaming tokenizer for JSON or similar text-based formats, but may alternatively be implemented as a schema-driven deserializer, memory-mapped accessor, database cursor, or other mechanism appropriate to the data format.

[0028] The execution module 130 receives the fused executable program 112 and processes data elements 122 to produce rule decisions 106. The execution module maintains rule state 132 indicating which rules have been resolved (true or false) and which remain pending. The execution module uses a selector dispatch mechanism 134 to efficiently route data elements to relevant instructions. When all rules have been resolved, the execution module signals early exit 136 to terminate evaluation before consuming remaining input.

[0029] The execution module may be embodied as a bytecode virtual machine interpreting bytecode instructions, a native code executor running JIT-compiled or AOT-compiled machine code, a graph executor traversing a decision graph structure, or other execution mechanisms. The choice of execution mechanism is an implementation detail that does not affect the fundamental operation of selector-based dispatch, predicate fusion, and early exit.

Compiler Architecture

[0030] Referring now to FIG. 2, the compilation process 200 transforms a set of independent rule definitions into a single fused executable program through a multi-phase pipeline.

[0031] In the parsing phase 210, each rule definition is parsed from its source representation (e.g., a domain-specific language, JSON configuration, or programmatic API) into a normalized internal representation. Each rule is decomposed into selector-predicate pairs, where a selector identifies a path within the structured data (e.g., "$.user.id" in JSONPath notation) and a predicate specifies a condition to evaluate against the value at that path (e.g., "equals 'admin'" or "greater than 1000").

[0032] As used herein, "path" and "selector" refer broadly to any identifier or address that locates a data element within structured data. This includes hierarchical paths in nested structures (e.g., "$.user.address.city" in JSON), but also encompasses flat identifiers such as column names in database rows, keys in key-value stores, field names in flat records, tuple positions, and any other addressing mechanism that identifies a specific data element within a larger data structure. The selector deduplication and fusion techniques described herein apply regardless of whether the underlying data model is hierarchical, relational, or flat.

[0033] In the normalization phase 220, selectors are converted to a canonical form. Path expressions are decomposed into sequences of path components (keys, array indices, wildcards, column names, or tuple positions as appropriate to the data format). Predicates are normalized to canonical operators and operand representations.

[0034] In the selector deduplication phase 230, all unique selectors across all rules are collected and organized into a selector data structure. In a preferred embodiment for hierarchical data, this structure is a selector prefix tree (trie). Referring to FIG. 3, the selector trie 300 has a root node 310 representing the document root. Each edge represents a path component (key name or array accessor), and each node represents a unique path prefix. Leaf nodes and intermediate nodes may have associated predicate lists indicating rules that require evaluation at that path. For flat data structures, the selector data structure may be a hash table or array mapping selectors directly to predicate lists.

[0035] For example, if three rules reference paths "$.user.id", "$.user.name", and "$.user.id" (duplicate), the selector trie would contain a path from root through "user" to both "id" and "name" nodes, with the "id" node annotated with predicates from both rules that reference it. This deduplication ensures that navigating to "$.user.id" in the input data triggers a single traversal, with the resulting value broadcast to all dependent predicates.

[0036] The value broadcast mechanism is a central feature of the fused execution model. When a value is extracted at a deduplicated selector, that value is made available to all predicates associated with that selector without re-extraction or re-traversal. This broadcast-once, evaluate-many approach is what enables evaluation cost to remain independent of rule count: adding more rules that reference the same selector incurs zero additional extraction

cost, with only the marginal cost of evaluating the additional predicates against the already-extracted value.

[0037] In the fusion phase 240, the selector data structure is traversed to generate a fused executable program. For each unique selector, instructions are emitted to: (a) match the selector against incoming data elements; (b) evaluate all predicates associated with that selector using the extracted value; and (c) update the rule state for each affected rule.

[0038] In the optimization phase 250, standard compiler optimizations are applied, including: constant folding for predicates with literal operands; dead code elimination for unreachable branches; branch reordering to maximize early exit probability; and common subexpression elimination for shared predicate computations.

[0039] The output of the compiler is a fused executable program 260 comprising: a selector dispatch table mapping selectors to instruction offsets or entry points; an instruction sequence or execution graph; and metadata including rule count, selector count, and memory bounds.

Bytecode Virtual Machine Embodiment

[0040] In a preferred embodiment, the fused executable program is a bytecode program interpreted by a virtual machine. Referring now to FIG. 4, the bytecode instruction set 400 is designed for efficient interpretation during streaming evaluation. Instructions are encoded in a compact binary format with fixed-width opcodes and variable-width operands.

[0041] Each instruction comprises a one-byte opcode followed by zero or more operands. Operands include rule indices (identifying specific rules in the rule state bitmap), selector identifiers (referencing entries in the selector dispatch table), constant identifiers (referencing entries in a constant pool), and branch offsets (for conditional execution). Operands are encoded using length-prefixed or fixed-width formats depending on the maximum value range determined at compile time. Execution proceeds sequentially through the instruction stream unless altered by branch instructions, ensuring deterministic behavior.

[0042] The instruction set includes the following representative instructions:

MATCH_PATH: Compares the current data element path against a specified pattern and branches based on match result.

CHECK_EXISTS: Sets a rule to true or false based on whether a value exists at the matched path.

CHECK_EQ: Compares the current element value against a constant and sets rule state based on equality.

CHECK_LT, CHECK_GT, CHECK_LE, CHECK_GE: Numeric comparison instructions.

CHECK_TYPE: Verifies that a value is of a specified type (string, number, boolean, null, object, array).

SET_RULE_TRUE: Marks a specified rule as resolved with value true.

SET_RULE_FALSE: Marks a specified rule as resolved with value false.

EARLY_EXIT: Checks whether all rules are resolved and terminates evaluation if so.

HALT: Terminates evaluation and returns current rule states.

[0043] The bytecode format is versioned to support evolution of the instruction set. The program header includes a version identifier and feature flags indicating which instruction set extensions are used.

Streaming Tokenizer Embodiment

[0044] In a preferred embodiment for text-based structured data such as JSON, the data element provider 120 is implemented as a streaming tokenizer. Unlike conventional parsers that construct complete parse trees or object graphs, the streaming tokenizer produces data elements incrementally as input bytes are received.

[0045] For JSON input, the tokenizer emits the following element types: START_OBJECT, END_OBJECT, START_ARRAY, END_ARRAY, KEY (object key names), and VALUE (strings, numbers, booleans, null). Each element is accompanied by path context including the current nesting depth and, for object contexts, the key name leading to the current position.

[0046] The tokenizer implements zero-copy semantics where string and numeric values are represented as slices referencing the original input buffer rather than allocated copies. This eliminates memory allocation during evaluation and enables cache-efficient processing.

[0047] The tokenizer maintains minimal state: a depth counter, a stack of context frames (object vs. array at each level), and continuation state for handling values that span input chunk boundaries. The maximum stack depth is configurable, providing a compile-time memory bound.

[0048] The tokenizer performs structural validation, detecting malformed input such as mismatched brackets, invalid escape sequences, or illegal character sequences. Upon detecting malformed input, the tokenizer transitions to an error state and emits an error element, enabling the execution module to implement fail-closed semantics.

Alternative Data Element Provider Embodiments

[0049] The data element provider may be implemented using mechanisms other than streaming tokenization depending on the data format and deployment context.

[0050] For schema-driven binary formats such as Protocol Buffers, Apache Avro, FlatBuffers, or Cap'n Proto, the data element provider is a deserializer that traverses the binary structure according to the schema, emitting data elements with path context as fields are encountered. This approach eliminates parsing overhead for formats with pre-defined schemas.

[0051] For memory-mapped data structures, the data element provider traverses the in-memory structure directly, emitting path-annotated elements without serialization or deserialization overhead. This is applicable to scenarios where data is already resident in memory in a structured form.

[0052] For database row streams, the data element provider wraps a database cursor, emitting each column value as a data element with the column name as path context. This enables fused rule evaluation over query result sets.

[0053] For network packets, the data element provider parses packet headers and payloads according to protocol specifications, emitting fields as data elements. This enables deep packet inspection with fused multi-rule evaluation.

Selector-First Execution Model

[0054] Referring now to FIG. 5, the fused semantic execution engine employs a selector-first execution model 500 that inverts the conventional rule-first approach.

[0055] In conventional rule-first evaluation, the outer loop iterates over rules, and for each rule, the system traverses the data to find relevant values. This results in repeated data traversal proportional to rule count.

[0056] In selector-first evaluation, the outer loop processes data elements from the data element provider. For each data element, the system performs O(1) dispatch 510 to determine which instructions (if any) are relevant to the current path. The selector data structure constructed during compilation is used to assign unique selector identifiers to each distinct path pattern. At runtime, dispatch is implemented using these precomputed selector identifiers, which map directly to instruction offsets or entry points via an index-based or hash-based lookup table. This two-phase approach—compile-time structure construction followed by runtime identifier-based dispatch—ensures constant-time path-to-instruction resolution regardless of the number of rules or selectors.

[0057] When a data element matches a selector, the associated instructions are executed 520. These instructions evaluate predicates and update rule state. Because selectors are deduplicated during compilation, a single value extraction serves all rules that reference the same path, with the extracted value broadcast to all dependent predicates without re-extraction.

[0058] When a data element does not match any selector in the dispatch table, the element is efficiently skipped 530 with minimal processing overhead. This selective processing ensures that evaluation cost is proportional to the number of matched selectors rather than total data size, providing substantial performance gains when rules are sparse relative to data content.

[0059] The selector-first model naturally supports wildcard selectors (e.g., "$.users[*].id" matching all array elements) through prefix-matching nodes in the selector data structure that activate on any array index or object key at the specified depth. Wildcard selectors match structurally at compile time; at runtime, a single wildcard node activates once per matching structural position in the input, with cost bounded by input structure rather than rule count.

Rule State and Early Exit Mechanism

[0060] Referring now to FIG. 6, the execution module maintains a rule state data structure 600 tracking the resolution status of each compiled rule. In the preferred embodiment, this structure is a fixed-size bitmap 610 allocated at compile time, with two bits per rule indicating: unresolved, resolved-true, or resolved-false.

[0061] As evaluation proceeds, predicate evaluations update the rule state. For conjunctive (AND) predicates, a single false predicate resolves the entire rule as false. For disjunctive (OR) predicates, a single true predicate resolves the rule as true. The compiler encodes these short-circuit semantics directly into the instruction sequence.

[0062] The early exit mechanism 620 monitors the rule state after each predicate evaluation. When all rules have transitioned from unresolved to either resolved-true or resolved-false, the mechanism triggers early termination 630, halting data element consumption and returning final rule states.

[0063] Early exit detection is implemented efficiently using a pending-rule counter that decrements as rules resolve. When the counter reaches zero, early exit is triggered. This avoids the overhead of scanning the entire bitmap after each update.

[0064] The early exit capability provides substantial performance benefits when input data contains sufficient information to resolve all rules before the complete data is received. In practice, API requests, log entries, and event messages frequently contain critical identifying information in early fields, enabling decisions before processing metadata, payloads, or trailing content.

Deterministic Fail-Closed Semantics

[0065] The fused semantic execution engine implements deterministic fail-closed semantics to ensure predictable behavior in security-critical applications.

[0066] Upon encountering malformed input (invalid syntax, encoding errors, structural violations), the data element provider emits an error element and transitions to an error state. The execution module responds by transitioning all unresolved rules to a configurable default state (typically false/deny), then terminating evaluation.

[0067] Upon encountering unexpected end-of-input (truncated data, connection termination), any unresolved rules are resolved to the default state. This ensures that incomplete data cannot result in spurious allow decisions.

[0068] The system avoids platform-dependent behavior that could introduce non-determinism. Numeric operations are performed using fixed-precision arithmetic or IEEE 754 with specified rounding modes. String comparisons use byte-wise comparison without locale-dependent collation. These choices ensure that evaluation produces identical results across different hardware platforms and operating systems.

[0069] Memory bounds are determined at compile time based on maximum selector depth, rule count, and configured buffer sizes. The system guarantees operation within these bounds without dynamic allocation during evaluation, preventing denial-of-service through memory exhaustion.

Performance Characteristics

[0070] The fused semantic execution engine achieves the following performance characteristics:

Time complexity with respect to input size: O(M), linear in the size of input data.

Time complexity with respect to rule count: independent of total rule count for rules sharing existing selectors.

Space complexity during evaluation: O(R + S), where R is rule count and S is maximum selector depth.

Compilation complexity: O(N) in the total size of rule definitions.

[0071] To clarify the rule-count independence: evaluation cost depends only on the number of unique selectors matched by the input data, not on the total number of compiled rules. The number of matched selectors is bounded by the structure of the input (i.e., the number of distinct paths present in the data) rather than by rule count. Since selector deduplication merges identical selectors across all rules during compilation, adding more rules that reference existing selectors incurs zero marginal extraction cost—only the marginal cost of evaluating additional predicates against already-extracted values. Adding rules that reference new selectors incurs cost proportional only to the new selector count, not the total rule count.

[0072] In empirical testing, the system has demonstrated evaluation rates of millions of events per second with hundreds of compiled rules, single-digit microsecond latency per evaluation, and throughput improvements of 5-16x compared to conventional sequential evaluation approaches, depending on rule count and selector overlap.

Exemplary Application Embodiments

[0073] The fused semantic execution engine may be applied to various domains requiring multi-rule evaluation over streaming structured data.

API Gateway Policy Enforcement

[0074] In an API gateway context, hundreds of policies governing authentication, authorization, rate limiting, request routing, and payload validation are compiled into a single fused program. Incoming HTTP requests with JSON bodies are evaluated in a single streaming pass, enabling sub-millisecond policy decisions regardless of policy count.

Real-Time Log Alerting

[0075] In a log processing context, thousands of alert rules from multiple tenants are compiled into a fused program. Log entries in JSON format are evaluated against all tenant rules in a single pass, enabling cost-effective multi-tenant alerting without per-tenant scanning overhead.

Financial Transaction Screening

[0076] In a financial context, transaction screening rules for fraud detection, sanctions compliance, and risk scoring are compiled into a fused program. Transaction messages are evaluated with deterministic fail-closed semantics ensuring that system failures result in transaction holds rather than approvals.

Blockchain Event Processing

[0077] In a blockchain monitoring context, rules for detecting arbitrage opportunities, liquidation events, or protocol-specific conditions are compiled into a fused program. Block and transaction data are evaluated with early exit enabling rapid detection of actionable events before complete block processing.

Additional Application Domains

[0078] The fused semantic execution engine is applicable to additional domains including but not limited to: network packet inspection and deep packet filtering wherein multiple firewall or intrusion detection rules are evaluated over packet streams; database query optimization wherein multiple query predicates are fused over streaming row data; schema validation wherein multiple structural and semantic constraints are evaluated during data parsing; machine learning feature extraction wherein multiple feature selectors are evaluated over input data in a single pass; content moderation and policy enforcement wherein multiple community guidelines or legal requirements are evaluated against user-generated content; IoT and sensor stream processing wherein multiple threshold, pattern, and anomaly detection rules are evaluated over sensor readings; email and message filtering wherein multiple spam, phishing, and routing rules are evaluated per message; GraphQL and API query validation wherein multiple authorization, complexity, and rate-limiting constraints are evaluated per query; configuration validation wherein multiple policy and compliance rules are evaluated over infrastructure-as-code definitions; and data quality assessment wherein multiple validation rules are evaluated over data pipeline records.

Implementation Details

[0079] In the preferred embodiment, the fused semantic execution engine is implemented in the Rust programming language, leveraging Rust's ownership model for memory safety without garbage collection overhead. The implementation is suitable for deployment as a library linked into application code, as a standalone daemon process, or as a WebAssembly module for browser or edge deployment.

[0080] The system supports multiple input formats through pluggable data element provider implementations. The primary embodiment targets JSON, with the architecture supporting extension to other structured formats such as CBOR, MessagePack, Protocol Buffers, Avro, XML, and custom binary protocols through providers that emit compatible path-aware data elements.

[0081] In certain embodiments, optional SIMD (Single Instruction, Multiple Data) optimizations may be employed to accelerate common operations such as numeric comparisons and string matching on supported processor architectures.

[0082] In certain embodiments, the fused executable program may be compiled to native machine code using just-in-time (JIT) or ahead-of-time (AOT) compilation techniques, providing further performance improvements over bytecode interpretation at the cost of increased compilation time and platform-specific code generation.

CLAIMS

What is claimed is:

1. A computer-implemented method for evaluating a plurality of decision rules against streaming structured data, comprising:

   (a) receiving, by a compiler, a plurality of independent decision rules, wherein each rule comprises one or more selectors identifying locations within structured data and one or more predicates to evaluate against values at said locations;

   (b) compiling, by the compiler, said plurality of rules into a single fused executable program, wherein selectors common to multiple rules are deduplicated into a selector data structure and predicates are indexed by selector;

   (c) receiving structured data elements with associated location information during data ingestion without constructing an intermediate representation of the complete data;

   (d) for each data element, dispatching execution to instructions associated with a current data location using a selector dispatch mechanism;

   (e) evaluating predicates and updating a rule state data structure indicating resolution status of each rule; and

   (f) terminating evaluation when all rules have been resolved, wherein termination occurs before complete consumption of the input when sufficient information exists to resolve all rules.

2. The method of claim 1, wherein the selector data structure comprises a prefix tree having nodes representing path prefixes and edges representing path components.

3. The method of claim 1, wherein the selector dispatch mechanism performs O(1) dispatch from the current data location to relevant instructions using an index-based or hash-based lookup.

4. The method of claim 1, wherein time complexity of evaluation is O(M) with respect to input size M and independent of total rule count for rules sharing existing selectors.

5. The method of claim 1, wherein the fused executable program comprises bytecode instructions interpreted by a virtual machine.

6. The method of claim 1, wherein receiving structured data elements comprises receiving path-aware token events from a streaming tokenizer that parses text-based structured data.

7. The method of claim 6, wherein the streaming tokenizer implements zero-copy semantics wherein values are represented as slices referencing an input buffer without memory allocation.

8. The method of claim 1, further comprising implementing fail-closed semantics wherein upon encountering malformed or incomplete input, all unresolved rules are resolved to a default deny state.

9. The method of claim 1, wherein the rule state data structure comprises a fixed-size bitmap allocated at compile time with bits indicating unresolved, resolved-true, or resolved-false status for each rule.

10. The method of claim 5, wherein the bytecode instructions are selected from the group consisting of MATCH_PATH, CHECK_EXISTS, CHECK_EQ, CHECK_LT, CHECK_GT, SET_RULE_TRUE, SET_RULE_FALSE, EARLY_EXIT, and HALT.

11. The method of claim 1, wherein memory consumption during evaluation is bounded at compile time and no dynamic memory allocation occurs during evaluation.

12. The method of claim 1, wherein the structured data comprises JSON data.

13. The method of claim 1, wherein a value extracted at a deduplicated selector is broadcast to all predicates associated with that selector without re-extraction, such that adding rules referencing existing selectors incurs zero additional extraction cost.

14. The method of claim 1, wherein the fused executable program comprises native machine code produced by just-in-time or ahead-of-time compilation.

15. The method of claim 1, wherein receiving structured data elements comprises deserializing binary-encoded data according to a schema.

16. The method of claim 1, wherein the selectors comprise flat identifiers including at least one of column names, keys in key-value stores, field names in flat records, or tuple positions.

17. A system for fused semantic execution of multiple decision rules against streaming structured data, comprising:

a compiler module configured to receive a plurality of independent decision rules and produce a single fused executable program, wherein the compiler deduplicates selectors common to multiple rules into a selector data structure and indexes predicates by selector;

a data element provider configured to provide structured data elements with associated location information during data ingestion without constructing an intermediate representation of the complete data;

an execution module configured to execute the fused executable program, receive data elements from the data element provider, and maintain a rule state data structure indicating resolution status of each rule; and

an early exit mechanism configured to monitor the rule state data structure and terminate evaluation when all rules have been resolved, before complete consumption of the input.

18. The system of claim 17, wherein the execution module comprises a virtual machine configured to interpret bytecode instructions.

19. The system of claim 17, wherein the data element provider comprises a streaming tokenizer configured to parse text-based structured data and emit path-aware token events.

20. The system of claim 17, wherein the selector data structure comprises a prefix tree and the execution module performs O(1) dispatch from current location to relevant instructions using an index-based or hash-based lookup derived from precomputed selector identifiers.

21. The system of claim 17, wherein evaluation time complexity is O(M) with respect to input size and independent of total rule count for rules sharing existing selectors.

22. The system of claim 17, further comprising fail-closed logic configured to resolve all unresolved rules to a default state upon encountering malformed or incomplete input.

23. A non-transitory computer-readable medium storing instructions that, when executed by one or more processors, cause the one or more processors to:

compile a plurality of independent decision rules into a single fused executable program, wherein selectors common to multiple rules are deduplicated and predicates are indexed by selector;

receive structured data elements with associated location information during data ingestion without constructing an intermediate representation of the complete data;

for each data element, dispatch execution to instructions based on current data location;

evaluate predicates and update rule resolution state; and

terminate evaluation when all rules are resolved, before complete consumption of input when sufficient information exists.

24. The non-transitory computer-readable medium of claim 23, wherein evaluation time complexity is O(M) with respect to input size and independent of total rule count for rules sharing existing selectors.

25. The non-transitory computer-readable medium of claim 23, wherein memory consumption during evaluation is bounded at compile time with no dynamic memory allocation during evaluation.

26. The non-transitory computer-readable medium of claim 23, wherein the instructions further cause the one or more processors to implement fail-closed semantics resolving unresolved rules to a default state upon encountering malformed or incomplete input.

ABSTRACT

A fused semantic execution engine for evaluating multiple independent decision rules against streaming structured data in a single pass. The engine comprises a compiler that transforms rule definitions into a fused executable program with deduplicated selectors and indexed predicates, a data element provider that supplies structured data elements with location context during data ingestion without constructing intermediate representations, and an execution module that processes instructions using selector-first dispatch and maintains rule resolution state. Values extracted at deduplicated selectors are broadcast to all dependent predicates without re-extraction. An early exit mechanism terminates evaluation when all rules are resolved, before complete input consumption. The system achieves O(M) time complexity with respect to input size and evaluation cost independent of total rule count for rules sharing existing selectors, with bounded memory consumption determined at compile time and deterministic fail-closed semantics on malformed or incomplete input. Selectors may identify hierarchical paths, flat identifiers such as column names or keys, or other data location mechanisms.