Greggle & Gruggle: Composable Regular Path Queries and Graph Manipulation

Sambuddha Majumder, Jayanta Majumder, Partha P Chakrabarti

January 25, 2026

Abstract

Greggle is a small query language and tool for performing regular path queries over labelled directed graphs. Gruggle is a companion Node.js utility for ingesting, merging, inspecting, and lightly manipulating graphs in the Graphviz `dot` format. Graphviz is a widely used system for graph visualization; its `dot` language is simple to author and makes it easy to view results with standard Graphviz tools. Together the two utilities provide a frictionless workflow: Gruggle builds, merges, filters, and styles graphs; Greggle answers expressive path queries with edge-level predicates; and Gruggle can consume Greggle's annotations (e.g., `find-path`) to visualize witnesses. This document presents both tools, why they are complementary, and how they can be used jointly in analysis and visualization tasks.

1 Introduction

Many applications, from network analysis and access-control policy checking to program analysis and model checking, involve reasoning about paths in a labelled directed graph. In such settings one often cares not merely about reachability, but about the *sequence of labels* seen along a path. Regular path queries provide a natural way to express such conditions.

Greggle is a lightweight system for expressing and evaluating such queries. Queries are S-expressions that combine regular expressions over edge labels with logical connectives (AND, OR, NOT) and existential quantification over node variables. A particular emphasis of Greggle is on *edge-level predicates*: small boolean formulas over labels that must hold on individual edges, integrated into the regular expressions. Internally, query evaluation is performed using BDD-backed finite-domain relations [1], but this document focuses on surface behaviour and usage. The automata-based path interrogation approach follows classical constructions [2].

Gruggle is a small Node.js tool for ingesting, merging, inspecting, and lightly manipulating graphs in the Graphviz `dot` format from the command line. It combines a permissive `dot` parser with a composable subcommand pipeline that can add nodes, add edges (optionally random), prune isolated nodes, perform path-based keep/remove filtering, and export to SVG via Graphviz `dot`.

Together, Gruggle provides practical graph construction, merging, selection, and visualization; Greggle provides formal path-query answering. Gruggle can call Greggle's `find-path` to mark a witness path with a label, and Greggle can operate on Gruggle-produced dot files. The combined workflow lets you script graph edits, run precise path queries, and visualize results, all with small, dependency-light tools.

2 Toolchain Overview

- **Build and shape graphs with Gruggle.** Parse and merge *dot* formatted graphs, add synthetic structure (chains, rings, grids, cliques), delete or filter nodes/edges, and style elements.

- **Ask path questions with Greggle.** Express regular-path conditions with edge predicates (`all-of`, `some-of`, `negate`) and logical composition over node variables.
- **Visualize and iterate.** Use Gruggle to emit SVGs of intermediate or final graphs. Invoke `find-path` to have Greggle annotate one matching path for inspection.
- **Directed analytics.** Gruggle offers indegree/outdegree selection; Greggle specialises in directed labelled paths, together covering common directed-graph tasks such as traffic-flow analysis or policy auditing.

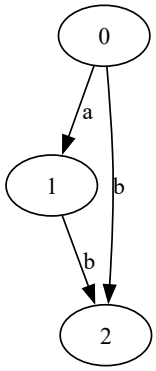
3 Greggle: Regular Path Queries with Edge Predicates

We now include the Greggle specification, highlighting how it can consume dot formatted graphs (including those shaped by Gruggle) and how its `find-path` output can be re-imported into Gruggle for visualization.

3.1 Graphs

A graph is a finite labelled directed graph $G = (V, E)$, where V is a set of nodes and $E \subseteq V \times 2^{\Sigma} \times V$ is a set of edges. Each edge (u, L, v) goes from node u to node v and carries a finite set of labels $L \subseteq \Sigma$ from a fixed alphabet of strings Σ.

In our prototype implementation and examples we often use Graphviz dot snippets to describe graphs, but the paradigm applies equally well to any in-memory labelled graph structure or object model. For example Figure 1 defines a graph with three nodes and three labelled edges. The dot parser supports multi-labelled edges by allowing comma-separated labels in the attribute value, e.g. `label="a,b"` denotes $L = \{\mathtt{a}, \mathtt{b}\}$.




```
digraph neg_demo {
  0;
  1;
  2;
  0 -> 1 [label="a"];
  0 -> 2 [label="b"];
  1 -> 2 [label="b"];
}
```

Figure 1: An edge-labelled graph and its dot format specification

3.2 Edge Predicates

At the heart of Greggle's regular expressions are *edge-level predicates*. Intuitively, an edge predicate P is a boolean expression that can be evaluated on the label set L of a single edge, yielding true or false. Greggle provides the following primitive constructors:

- Atomic label test: `a` is true on an edge whose label set contains the symbol `a`.

- Wildcard: `dot` is true on every edge, regardless of its label set (analogous to "." in traditional regular expressions).
- Negation: `(negate` P`)` is true when P is false on that edge.
- All-of: `(all-of` P_1 P_2 `...)` is true when each P_i is true on that edge.
- Some-of: `(some-of` P_1 P_2 `...)` is true when at least one P_i is true on that edge.

These constructors can be nested arbitrarily. As convenient shorthand, a leading tilde on an atom is interpreted as negation, so `~a` is equivalent to `(negate a)`.

For example:

- `a` holds on an edge with labels $\{\texttt{a}, \texttt{b}\}$; `~a` does not.
- `(all-of a b)` holds on an edge with labels $\{\texttt{a}, \texttt{b}\}$ but not on one with just $\{\texttt{a}\}$.
- `(some-of b c)` holds on an edge with labels $\{\texttt{b}, \texttt{c}\}$ and also on one with labels $\{\texttt{b}\}$ or $\{\texttt{c}\}$.

3.3 Path Expressions

Path expressions describe sets of finite paths in the graph. They are regular expressions whose atomic symbols are edge predicates. The syntax is:

- `P`, where P is an edge predicate, denotes a single edge on which P holds.
- `(concat` r_1 r_2 ... r_n`)` denotes concatenation: paths formed by following a subpath in r_1, then r_2, etc.
- `(alt` r_1 r_2 ... r_n`)` denotes alternation: a path is accepted if it is accepted by some r_i.
- `(star` r`)` denotes Kleene star, any finite concatenation of paths in the language of r (including the empty path).
- `(plus` r`)` denotes one-or-more repetitions of r.

In practice, an atomic symbol `a` is treated as the edge predicate testing for label `a`, and we omit the explicit `sym` constructor sometimes found in other presentations.

3.4 Query Expressions

Building on path expressions, Greggle defines expressions that relate node variables:

- `(exists-path x y` r`)` is true under an assignment of nodes to variables x and y when there exists a path from x to y in the graph whose sequence of edge labels is accepted by the path expression r.
- `(match x` p`)` is true when the label of node x matches the string regular expression p in the usual sense of `std::regex_match`.
- `(no-edge x y)` is true when there is *no* edge from x to y in the graph.
- `(no-connection x y)` is true when there is no edge between x and y in either direction (neither $x \to y$ nor $y \to x$).
- `(same x y)` is true when x and y are assigned the same node.

- `(different x y)` is true when x and y are assigned different nodes.
- `(and` e_1 e_2 `...` e_n`)` is true when all subexpressions are true under the current assignment.
- `(or` e_1 e_2 `...` e_n`)` is true when at least one e_i is true.
- `(not` e`)` is the logical complement.
- `(exists (x y ...)` e`)` existentially quantifies over the listed variables, projecting them out of the final result.

The free variables of an expression are those that are not bound by an enclosing `exists`. Evaluating a query on a fixed graph yields a relation over the free variables; the query tool displays this as a set of satisfying bindings.

4 Informal Semantics

4.1 Path Semantics

Path expressions are interpreted as sets of paths over labelled edges. For `exists-path`, this induces a binary relation over nodes: (u, v) is in the relation when there is a path from u to v whose label-sequence is accepted by the path expression. Alternation yields union of relations, concatenation yields relational composition, star yields reflexive-transitive closure, and plus yields transitive closure.

4.2 Expression Semantics

Queries over variables compose relations using standard boolean and relational connectives. For example, `(and (exists-path x y r1) (exists-path y z r2))` denotes the set of triples (x, y, z) where both relations hold. Projection via `exists` removes bound variables from the reported relation; `not` and the binary absence tests allow expressing negative conditions (e.g., "no edge between `x` and `y`").

5 Implementation Sketch

The implementation follows the usual automata-based approach to regular path queries but adds an explicit predicate evaluator on edges and a symbolic relation layer:

1. **NFA construction:** Each path expression is compiled to an NFA with predicate-labelled transitions; concatenation, alternation, star, and plus are handled via epsilon-transitions in the standard way.
2. **Product search:** For `exists-path`, the NFA is run over the graph by BFS over pairs of (graph node, NFA state), enabling transitions when the edge predicate holds on the edge's label set.
3. **Relational composition:** The relations produced by `exists-path` are combined using relational operators (intersection, union, projection, complement), implemented symbolically using BDD-backed finite-domain relations.

Path Evaluation via NFA

Each path expression is compiled into a small NFA. States are classified as accepting or non-accepting, and transitions are labelled either by edge predicates (for consuming one edge) or by epsilon-transitions (for moving between states without consuming an edge). Regular expression constructs are handled in the standard way:

- A symbol (edge predicate) becomes a two-state NFA with a single consuming transition.
- Concatenation connects the end states of each component using epsilon-transitions.
- Alternation introduces a fresh start and end state and connects them via epsilon-transitions to and from each component.
- Star and plus introduce loops via epsilon-transitions to allow repeated traversal.

To evaluate `(exists-path x y` r`)` on a graph G, the system runs a breadth-first search over pairs (u, q) where u is a graph node and q is an NFA state. For each start node s, the search is initialized at all NFA states reachable from the NFA start by epsilon-transitions, paired with s. The algorithm repeatedly:

- Dequeues a pair (u, q).
- If q is accepting, records $(x = u, y = u')$ as satisfying, where u' is the current node in that pair.
- For each outgoing edge (u, L, v) in G and each NFA transition from q labelled by edge predicate P, checks whether P holds on the label set L. If so, it follows that transition to a new NFA state q', computes the epsilon-closure of q', and enqueues any new pairs (v, q'') not seen before.

The result of this search is a set of node pairs (s, t) that constitute the relation for `exists-path` over the variables (x, y).

Symbolic Relations with BDDs

Relations over node variables are represented symbolically using a finite-domain BDD layer. Each logical variable is associated with a finite-domain identifier; the underlying BDD package provides operations to represent and manipulate finite-domain predicates over these identifiers. A relation is thus encoded as a BDD whose variables correspond to the finite-domain variables for the logical variables.

Given a tuple of node values $(v_1, \ldots, v_k)$ over variables $(x_1, \ldots, x_k)$, the system constructs a conjunction of finite-domain predicates stating that each variable x_i equals value v_i, and ORs these into the BDD representing the relation. From this representation, the following relational operations are implemented:

- **Union** and **Intersection** via BDD disjunction and conjunction, respectively.
- **Set difference** by conjunction with the negation of the other relation.
- **Projection** by existential quantification over the finite-domain variables corresponding to quantified logical variables.
- **Complement** by BDD negation, with respect to the full product domain of the participating variables.

Edge Predicate Evaluation

Edge predicates are represented as small trees with nodes for atomic tests and boolean connectives (negation, all-of, some-of). Each atomic test stores a label name. Evaluation of a predicate P on an edge is implemented by recursively traversing this tree:

- An atomic predicate compares its label name to the edge's label set.
- A negation node inverts the result of its child.
- An all-of node checks that all child predicates hold.
- A some-of node checks that at least one child predicate holds.

During NFA traversal, these predicates are used to decide which transitions are enabled on a given edge. This design cleanly separates path structure (handled by the NFA) from local label conditions (handled by predicate evaluation).

Query Tool Pipeline

The command-line query tool ties these pieces together:

1. Parses a dot snippet or other graph description into the internal graph structure.
2. Parses the query S-expression into the expression and path ASTs, including edge predicates.
3. Evaluates the expression on the graph, constructing and combining symbolic relations as needed.
4. Traverses the resulting relation and prints each satisfying binding, mapping internal node indices back to external node identifiers where appropriate.

This architecture allows Greggle to support expressive regular path queries with edge-level predicates while keeping the implementation modular: path processing is localized to the NFA construction and product search, whereas relational composition and quantification are handled uniformly by the BDD-backed finite-domain layer.

6 Examples

6.1 An Example Graph

Figure 2 shows an example graph with mixed labels, illustrating the style of graphs Greggle is designed to work with. Nodes represent network states, edges represent transitions labelled by actions or propositions.

On this graph, one can express queries such as:

- Paths starting with `a` and then repeating `b a` one or more times:

```
(exists-path x y (concat a (plus (concat b a))))
```

- Pairs of nodes connected by a path of only `c`-edges and also by a path matching the previous pattern:

```
(and (exists-path x y (star c))
     (exists-path x y (concat a (plus (concat b a)))))
```

These examples exercise nested regular expressions and relational conjunction.

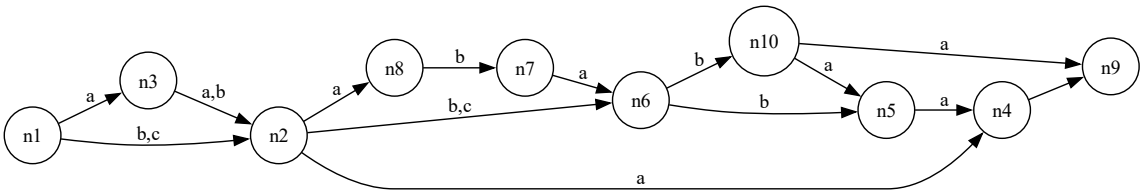


Figure 2: Example labelled graph used in several path queries.

6.2 Negated Edge Predicate

Consider again the small graph `neg_demo` mentioned in Section 3.1. The query

```
(exists-path x y ~a)
```

asks for all pairs (x, y) such that there exists a single edge from x to y whose label set does not contain `a`. On this graph, the satisfying bindings are:

```
Satisfying bindings:
x=0 y=2
x=1 y=2
```

corresponding exactly to the two edges labelled `b`.

6.3 All-of and Some-of

Now consider the graph in Figure 3.

```
digraph all_some_demo {
  0;
  1;
  2;
  0 -> 1 [label="a,b"];
  0 -> 2 [label="a"];
  1 -> 2 [label="b,c"];
}
```

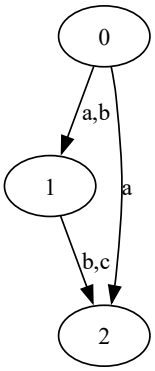


Figure 3: A graph with multi-labelled edges

The query

```
(exists-path x y (all-of a b))
```

asks for all edges that have both labels `a` and `b`. On this graph, the only such edge is $0 \to 1$, and the result is:

```
Satisfying bindings:
x=0 y=1
```

The query

```
(exists-path x y (some-of b c))
```

asks for all edges that have `b` or `c` (or both). Here both $0 \to 1$ and $1 \to 2$ satisfy this predicate, yielding:

```
Satisfying bindings:
x=0 y=1
x=1 y=2
```

These examples demonstrate how edge-level predicates express conditions involving the presence or absence of multiple labels on a single edge.

6.4 Wildcard and Relational Predicates

The `dot` predicate acts as a wildcard edge test: it is true on every edge, regardless of its label set. On the graph `neg_demo` from Section 3.1, the query

```
(exists-path x y dot)
```

asks for all single-edge connections in the graph, and yields exactly the set of edges:

```
Satisfying bindings:
x=0 y=1
x=0 y=2
x=1 y=2
```

The predicate `(no-edge x y)` expresses the absence of a directed edge from x to y, while `(no-connection x y)` expresses the absence of any edge between x and y in either direction. On `neg_demo`, for example, `(no-connection x y)` holds only when x and y are the same node, since every unordered pair of distinct nodes is joined by at least one edge.

Finally, the predicates `(same x y)` and `(different x y)` implement equality and inequality over node variables. On a graph with nodes $0, 1, 2$, `(same x y)` yields bindings `x=0,y=0`, `x=1,y=1`, `x=2,y=2`, while `(different x y)` yields all pairs where the node components are distinct. These relational primitives make it easy to combine path-based conditions with simple structural constraints on node assignments.

As a final illustration, consider a graph whose node labels are strings such as `A`, `AB`, `B1`, `Z`, with edges labelled by atomic propositions `a`, `b`. The query

```
(and (match x "A.*")
     (exists-path x y a))
```

asks for all pairs (x, y) such that the label of x matches the pattern `A.*` and there is an edge labelled `a` from x to y. On a graph where there is an `a`-edge from `A` to `Z` but only a `b`-edge from `AB` to `Z`, the only satisfying binding is `x=A,y=Z`, combining a label-based filter with a path predicate.

7 DOT-Based Query Tool

The Greggle query tool is a command-line program that takes a Graphviz dot file and a query expression as arguments, and prints the satisfying bindings. Its usage is:

```
greggle <graph.dot> "<query in lisp syntax>"
```

where the query must be quoted so that it is passed as a single argument.

Internally, the tool:

1. Parses the Graphviz dot file, assigning each node identifier a unique integer index and collecting edge label sets.
2. Parses the query S-expression, building the abstract syntax tree for path and edge predicates.
3. Evaluates the query on the graph to obtain a relation over the free variables.
4. Prints each satisfying binding as a line of the form `x=nodeLabel y=nodeLabel`, mapping internal node indices back to the original Graphviz dot labels.

For example, running the tool on `neg_demo.dot` with the negated edge predicate query produces:

```
Satisfying bindings:
x=0 y=2
x=1 y=2
```

matching the hand analysis above.

8 Future Work

Possible extensions include incorporating richer node/edge attributes into predicates, adding shortest-path variants of regular path queries, supporting incremental updates for dynamic graphs, and tighter integration with visualization (e.g., highlighting all satisfying paths, not just one witness).

9 Gruggle: Graphviz Graph Manipulation and Visualization

9.1 Command Line and Entry Point

The main executable is `gruggle.js`. It accepts zero or more input files using repeated `-i` flags. All parsed graphs are merged; if no inputs are given an empty digraph is used. Style switches: `--curved` (default) for curved spline edges, `--ortho` or `--rectilinear` for rectilinear edges (`splines=ortho`). After parsing/merging, subcommands are applied in order. Examples:

```
node gruggle.js --ortho -i g1.dot -i g2.dot set-name merged to-svg merged.svg
node gruggle.js chain "add-node nx:1:100" "add-edge 50 nx.* nx.*" \
                    "keep-paths nx1 nx5" "to-svg pruned.svg"
```

9.2 Graphviz Dot-format Parsing and Merging

The parser handles both `graph`/`digraph`, optional `strict`, named/unnamed graphs, subgraphs, edge ops `->`/`--`, port/compass points, bracketed attribute lists, quoted IDs, HTML-like IDs, numbers, comments (`//`, `/* */`, `#`), and default attribute statements (`graph`, `node`, `edge`). When merging, later files' attributes override earlier ones on conflicts; edges are deduplicated by endpoints/ports/direction with attribute merge. Default emitted style: box-shaped, rounded, light-blue nodes and muted-blue edges; orthogonal vs. curved splines are controlled by the CLI switch above.

9.3 Subcommands

Subcommands operate on the in-memory graph and can be composed with the `chain` meta-command. Tokens inside a chain are parsed with a simple shell-like splitter (quotes supported).

- **set-name** `<name>`: rename the graph.
- **add-nodes** `<id>|<prefix:start:end>`: add a single node or a numeric range (e.g. `nx:1:3` → `nx1`, `nx2`, `nx3`).
- **add-edges** `<label> <fromRe> <toRe>`: add edges for all pairs whose IDs match the regexes, using the current graph direction. Attribute `label` is set when provided and non-empty.
- **add-edges** `<count> <fromRe> <toRe> [label]`: pick up to `count` random distinct pairs from the regex matches and add those edges (optional label).
- **add-chain** `<root> <n>`: add nodes `root_1`.. `root_n` linked in a chain.
- **add-grid** `<root> <rows> <cols>`: add a `rows` x `cols` grid with nodes `root_i_j` and directional grid edges (top-to-bottom, left-to-right).
- **delete-isolated-nodes**: remove nodes with no incident edges.
- **remove-nodes** `<regex>`: delete nodes matching the regex and their incident edges.
- **remove-edges-between-nodes** `<fromRe> <toRe>`: delete edges whose tail/head IDs match the regex pair.
- **add-labels-to-edges-on-paths** `<fromRe> <toRe> <label>`: add the given label on every edge lying on a path from sources matching `fromRe` to targets matching `toRe`.
- **remove-labels-from-edges-on-paths** `<fromRe> <toRe> <label>`: remove the given label from every edge lying on such paths; if no labels remain, the attribute is removed.
- **select-nodes** / **deselect-nodes** `<regex>`: mark/unmark nodes whose IDs match the regex.
- **select-edges** / **deselect-edges**: either `<labelRegex>` (matches any label component) or `<fromRe> <toRe>` (matches endpoints) to mark/unmark edges.
- **select-nodes-label** `<labelRegex>`: select nodes whose label attribute has a matching component.
- **delete-selected**: delete selected nodes (and their incident edges) and selected edges.
- **set-node-attr-on-selected** `<attr> <value>` / **set-edge-attr-on-selected** `<attr> <value>`: set a dot attribute on selected nodes/edges.

- **unset-node-attr-on-selected** <attr> / **unset-edge-attr-on-selected** <attr>: remove a dot attribute on selected nodes/edges.
- **clear-visual-attrs-on-selected**: clear common visual attributes (color, fillcolor, fontcolor, style, shape, penwidth, arrowsize, fontsize) on selected nodes/edges.
- **add-labels-to-selected** <label>: append the label (comma-separated) to all selected nodes and edges.
- **remove-labels-from-selected** <label>: remove the label from selected nodes/edges; drops the attribute if empty.
- **keep-paths** <fromRe> <toRe>: mark all nodes and edges that lie on at least one path from any source matching fromRe to any target matching toRe. Marks set keep=true and clear remove.
- **remove-paths** <fromRe> <toRe>: similarly mark nodes/edges on such paths, setting remove=true and clearing keep.
- **keep-path-edges** <fromRe> <toRe>: mark only the edges (not nodes) that lie on such paths with keep=true.
- **remove-path-edges** <fromRe> <toRe>: mark only the edges (not nodes) on such paths with remove=true.
- **filter-to-paths** <fromRe> <toRe>: keep only nodes/edges on such paths (mark all others as remove).
- **select-nodes-on-paths** <fromRe> <toRe>: select all nodes on such paths (for later bulk ops).
- **select-edges-on-paths** <fromRe> <toRe>: select all edges on such paths.
- **find-path** <regex> <label>: run Greggle find-path to annotate one matching path's edges with the given label.
- **select-neighbours** [in|out|both] <hops>: expand selection by BFS over neighbours (default both,1).
- **select-nodes-degree** <lt|le|eq|ge|gt> <k>: select nodes by total degree (in+out for digraphs).
- **select-nodes-indegree** <lt|le|eq|ge|gt> <k>: select nodes by indegree (undirected edges count as both).
- **select-nodes-outdegree** <lt|le|eq|ge|gt> <k>: select nodes by outdegree (undirected edges count as both).
- **select-component-of-nodes** <regex>: select nodes/edges in components of matching nodes.
- **select-components-with-selected**: select nodes/edges in any component that already contains a selected node.
- **clear-selection**: clear selection on all nodes/edges.
- **invert-selection-nodes**: flip selection flags on nodes.
- **invert-selection-edges**: flip selection flags on edges.
- **chain** "cmd1 ..." "cmd2 ...": run each nested command in order. Inside a chain, -f <file> splices commands line-by-line from the file (comments/blank lines skipped) at that point in sequence.
- **to-svg** <filename>: pipe the current graph through dot -Tsvg -o <filename> (assumes dot is on PATH).

Keep/remove semantics. Elements carry in-memory *keep* and *remove* flags that flip with the operations above (keep clears remove, remove clears keep). When emitting the final graph:

- If any element is marked keep, only kept elements are emitted.
- Otherwise, elements marked remove are suppressed; the rest are emitted.

No flags are written into dot attributes.

9.4 Output

Default output is dot on stdout, reflecting merged defaults and the current subcommand state. Using `to-svg` writes an SVG file via Graphviz with the same styling preamble (boxy light-blue nodes, blue edges, curved or orthogonal per CLI switch).

9.5 Example Workflow

```
# Merge two graphs, add 20 random edges among nx* nodes, retain only
# paths from nx1 to nx5, emit both dot and SVG with orthogonal edges.
node gruggle.js --ortho -i a.dot -i b.dot \
  chain "add-edge 20 nx.* nx.*" "keep-paths nx1 nx5" "to-svg pruned.svg" \
  > pruned.dot
```

9.6 Illustrative Examples

9.6.1 Ring/Chain/Grid Builders

Figure 4 and Figure 5 illustrate the synthetic builders discussed here.

```
# Chain of four nodes c_1 -> c_2 -> c_3 -> c_4
node gruggle.js chain "add-chain c 4" > chain.dot

# Ring of three nodes r_0 -> r_1 -> r_2 -> r_0
node gruggle.js chain "add-ring r 3 0" > ring.dot

# 2x2 grid g_1_1, g_1_2, g_2_1, g_2_2 with edges down/right
node gruggle.js chain "add-grid g 2 2" > grid.dot
```

9.6.2 Random Selection and Labeling

Figure 6 shows random node/edge selection annotated with `rnd`.

```
# Select up to 10 nodes and edges at random, mark them 'rnd'
node gruggle.js -i sample1.dot chain \
  "select-random-nodes 10 .*" \
  "select-random-edges 10 .* .*" \
  "add-labels-to-selected rnd" > random.dot
```

Resulting edges/nodes carry labels like `name:rnd` or `label="x,rnd"`.

9.6.3 Path Highlighting via Greggle (find-path)

Figure 7 shows greggle-driven path annotation.

```
# Highlight one path whose edge labels match (concat x y) with 'pathlab'
node gruggle.js -i sample1.dot chain \
  "find-path (concat x y) pathlab" > path_highlighted.dot
```

Edges on the witness path gain `pathlab` in their label (e.g., `label="x,pathlab"`).

9.6.4 Component-Based Selection

Figure 8 visualizes component selection from regex and from existing selection.

```
# Select components containing nodes matching x*
node gruggle.js -i comps.dot chain \
  "select-component-of-nodes x.*" \
  "add-labels-to-selected compA" > comps_marked.dot

# Select components containing already-selected nodes
node gruggle.js -i comps.dot chain \
  "select-nodes y2" \
  "select-components-with-selected" \
  "add-labels-to-selected compB" > comps2.dot
```

Nodes and internal edges in those components are labeled `name:compA` or `name:compB`.

9.6.5 Selection Utilities

Figure 9 shows styling and clearing of selected elements.

```
# Label-selected nodes/edges, then clear visual styling
node gruggle.js -i sample1.dot chain \
  "select-nodes s1" \
  "select-edges x" \
  "set-node-attr-on-selected fillcolor yellow" \
  "set-edge-attr-on-selected color red" \
  "clear-visual-attrs-on-selected" > styled.dot
```

9.6.6 Degree and Neighborhood Selection

Figure 10 illustrates degree-based selection and neighbour expansion before the corresponding command snippets.

```
# Select nodes with degree >= 3
node gruggle.js -i sample1.dot chain \
  "select-nodes-degree ge 3" \
  "add-labels-to-selected degsel" > deg.dot

# BFS expand selection by 2 hops (undirected sense)
node gruggle.js -i sample1.dot chain \
```

```
  "select-nodes s1" \
  "select-neighbours both 2" \
  "add-labels-to-selected neigh" > neigh.dot
```

9.6.7 Directed Indegree/Outdegree Selection

Figure 11 shows indegree and outdegree filtering on a directed sample graph.

```
# Indegree >= 2 (counts incoming arcs; undirected edges count as both)
node gruggle.js -i sample1.dot chain \
  "select-nodes-indegree ge 2" \
  "add-labels-to-selected indegsel" > indeg.dot

# Outdegree == 0 (no outgoing arcs; undirected edges count as both)
node gruggle.js -i sample1.dot chain \
  "select-nodes-outdegree eq 0" \
  "add-labels-to-selected outdegsel" > outdeg.dot
```

These directed-aware predicates complement the total-degree selector when working with asymmetric graphs.

10 Combined Use Cases

Policy and reachability audits. Use Gruggle to ingest firewall or service-topology dot, mark sources/targets, prune irrelevant nodes with `filter-to-paths`, then ask Greggle to find forbidden or required label sequences. Feed `find-path` results back into Gruggle to visualize witness or counterexample paths.

Dataflow debugging. Build SSA or CFG fragments with Gruggle (`add-chain`, `add-grid`, `add-edges`), then use Greggle to assert sequences of operations (e.g., taint propagation) with edge predicates like `(all-of read taint)`. Visualize offending paths with Gruggle.

Interactive exploration. Select dense regions with Gruggle's degree/indegree/outdegree selectors, expand neighbours, then run Greggle queries confined to those regions. This iterative loop narrows search space while preserving the ability to emit publication-ready diagrams.

11 Conclusion

Gruggle and Greggle form a lightweight, scriptable toolchain for labelled directed graphs: Gruggle for shaping, filtering, and rendering dot; Greggle for semantically rich regular path queries with edge predicates. Because both speak the dot format, they compose naturally: Gruggle prepares and visualizes; Greggle answers; Gruggle renders the answers. This combination supports quick exploratory analysis and repeatable pipelines without heavy dependencies.

References

[1] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[2] M. Y. Vardi. Automata-theoretic model checking revisited. In *Verification, Model Checking, and Abstract Interpretation*, pages 137–153. Springer, 2007.

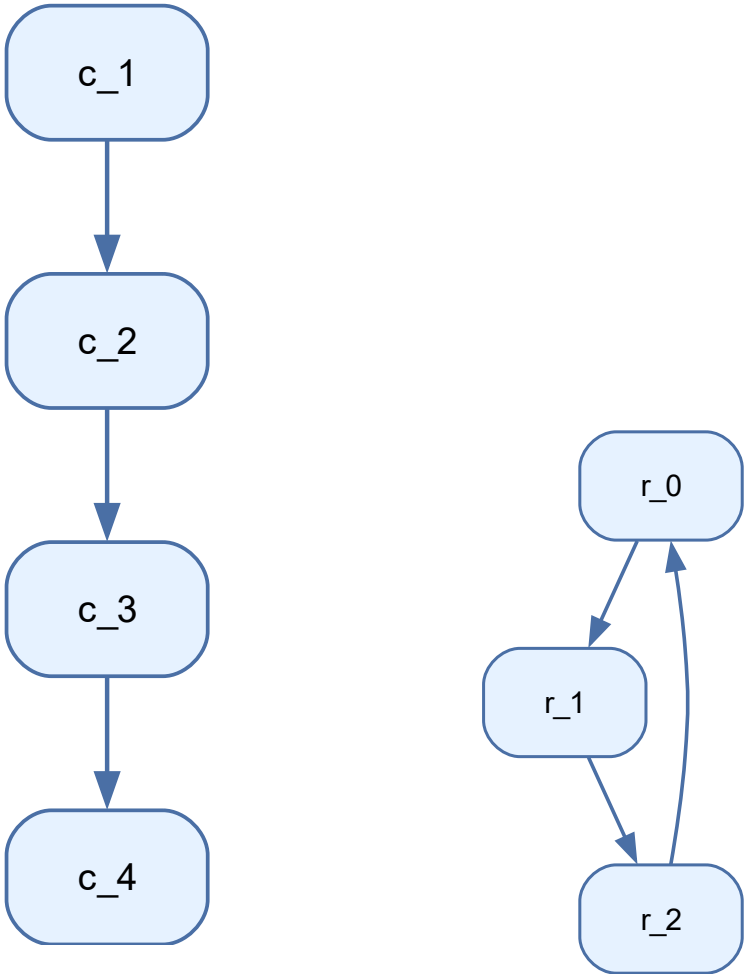


Figure 4: Left: chain of four nodes. Right: ring of three nodes.

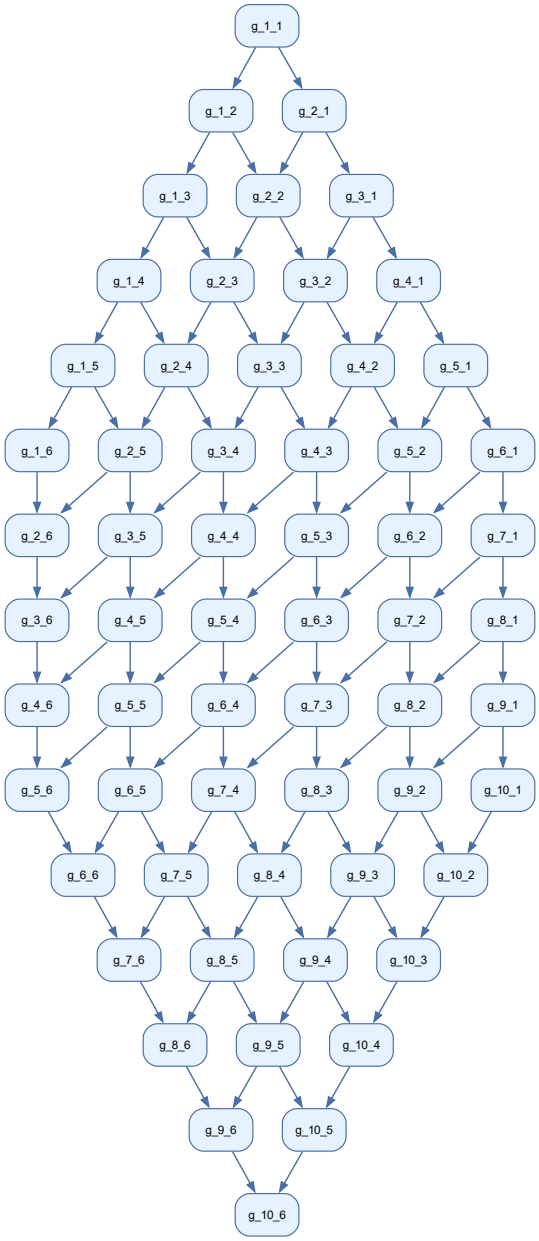


Figure 5: 2x2 grid built with `add-grid g 2 2`.

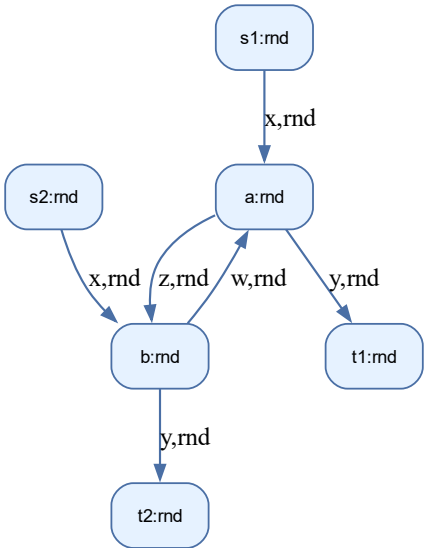


Figure 6: Randomly selected nodes/edges labeled `rnd`.

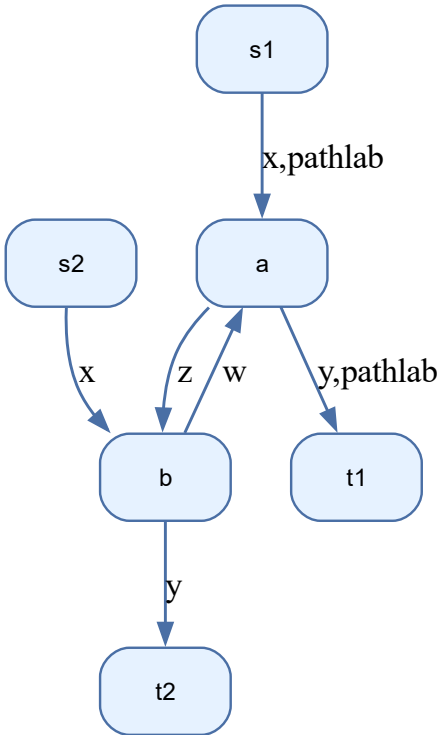


Figure 7: Edges on one matching path annotated with `pathlab`.

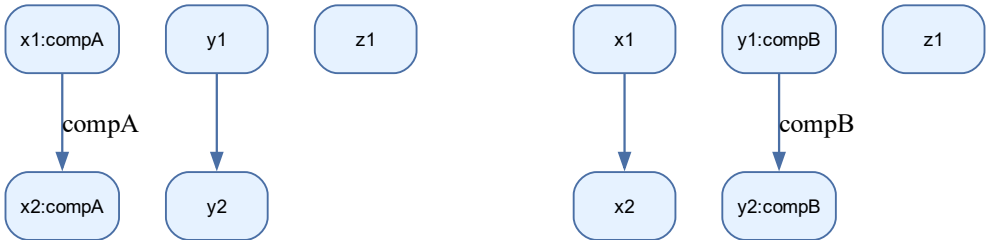


Figure 8: Selecting components by node regex (left) and from existing selection (right).

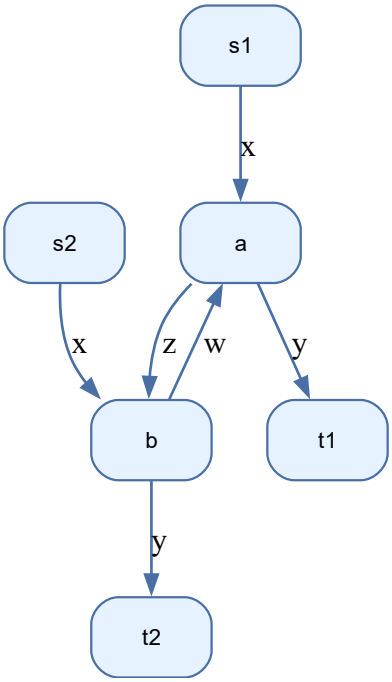


Figure 9: Visual attrs set then cleared on selected nodes/edges.

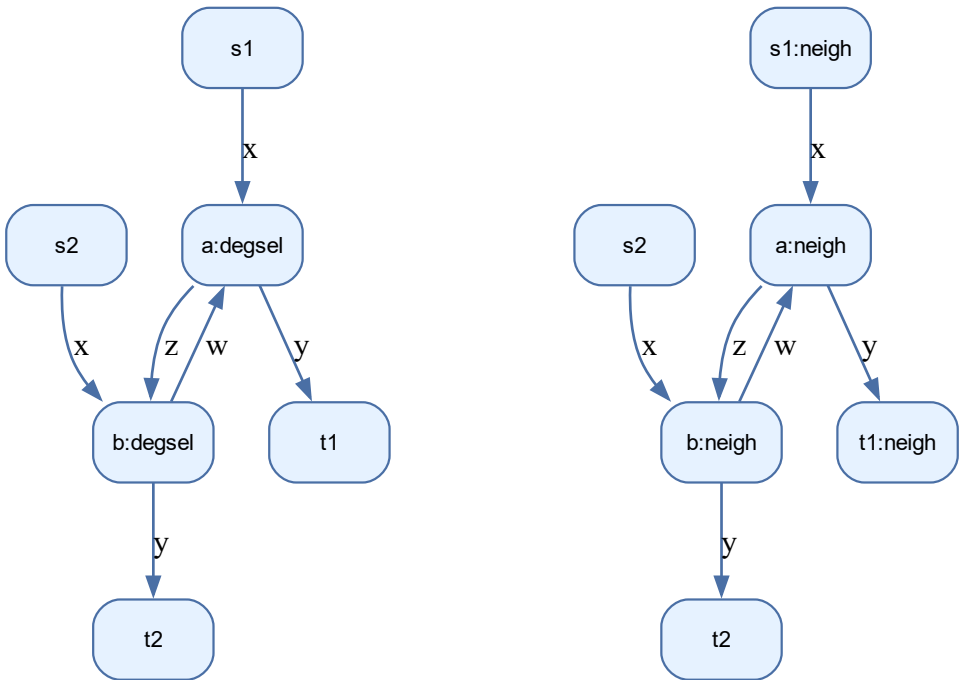


Figure 10: Left: nodes with degree ≥ 3 labeled `degsel`. Right: 2-hop neighbour expansion labeled `neigh`.

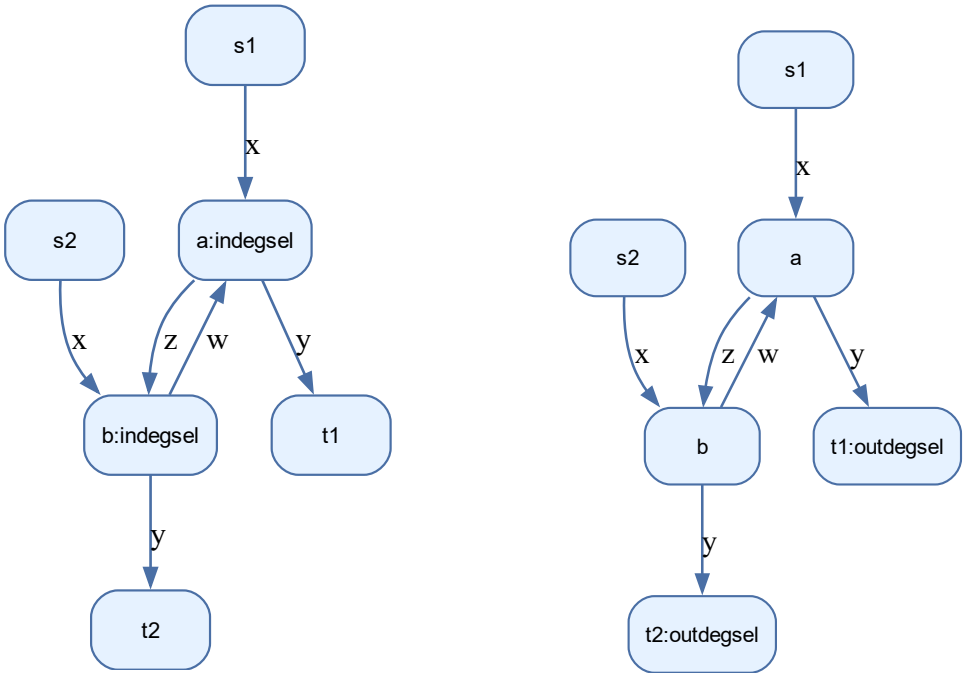


Figure 11: Left: nodes with indegree ≥ 2 labeled `indegsel`. Right: nodes with outdegree $= 0$ labeled `outdegsel`. Undirected edges count toward both in/out.