dstr: A Small Language for Systems That Become Graphs

Sambuddha Majumder Jayanta Majumder

May 24, 2026

Abstract

dstr is a compact language for describing finite dynamic systems in a form that is both easy to write and semantically explicit. Its primary interface is a small s-expression DSL in which one states variables, initial conditions, actions, invariants, and reachability goals without the verbosity that often discourages exploratory modeling. The resulting description is not intended merely to support a checker run. It is intended to generate an explicit state graph that can subsequently be queried, transformed, filtered, and published. The central claim of this paper is that a state-action language becomes substantially more valuable when its semantics are treated as a first-class graph artifact, suitable not only for validation but also for downstream graph analysis and presentation.

1 Introduction

The name dstr was chosen partly for its sound. It echoes the name Dijkstra, and in fact retains the non-silent consonants from that name. For what it is worth as an expansion, it also fits the phrase "describing states and transitions for reasoning". The name is therefore meant to suggest both a style of formal thought and the specific structural emphasis of the language.

Many important systems are easier to understand as spaces of possibilities than as streams of execution. A protocol, a controller, a workflow, or a concurrent routine is not merely a program that runs. It is a set of states, a family of allowed transitions, and a collection of claims we hope remain true throughout motion. Once viewed in that way, the natural mathematical object is a graph.

Yet graph construction is usually treated as a downstream detail. One first writes code, or a large formal model, or an ad hoc simulation, and only later attempts to recover a graph worth studying. The result is often unsatisfactory. Either the notation is pleasant but semantically loose, or the semantics are precise but the authoring burden is too high for exploratory use.

dstr begins from a different preference. The system author should be able to write down state and action structure in a small, expressive, tree-like surface language. That description should compile into a simple internal form, be checked by explicit exploration, and then be materialized as a graph that is valuable in its own right. The graph is not merely evidence that checking took place. It is a reusable object for later analysis.

2 The High-Level Interface

The high-level interface of dstr is now presented through two sibling surface languages: a Lisp-style DSL and a Tcl-shaped DSL called tdstr. Both compile to the same normalized JSON representation. The choice to support two front ends is not decorative. It separates semantic stability from authoring convenience. Once the core representation is fixed, one may admit multiple notations that are optimized for different deployment contexts while still preserving a single checker, a single graph-oriented workflow, and a single regression target.

The Lisp front end remains the most expressive meta-language for the system. This choice is not an affectation. S-expressions provide exactly the right compromise for a system description language:

- they are compact,
- they expose tree structure directly,
- they admit macros and normalization naturally, and
- they avoid the visual clutter of deeply tagged abstract syntax.

The aim is that one writes the system in a form close to its conceptual shape. State variables are named directly. Predicates appear as ordinary nested forms. Action structure is explicit without becoming ceremonial. The next-state view of a variable can be expressed tersely, and symbols can serve as literals where that is the most natural reading.

The DSL is also intentionally programmable. A specification file may define local macros and helper functions, or load a small library of such definitions, before stating the system itself. Ordinary expansion can be requested explicitly, and the current front end also supports compact macro shorthands in the surface notation. The implemented surface forms are precise. One may write `(expand macro-name ...)` for an explicit one-step expansion, `(%same x)` as shorthand for `(expand same x)`, and `(!unchanged x y)` when the macro expands to a list of sibling clauses to be spliced into the surrounding form. The predefined helper `same` expands to `(= x+ x)`, while `unchanged` expands to a list such as `((= x+ x) (= y+ y))`. The front end also accepts top-level `defmacro`, `defun`, and `load` forms before the `system` form, so a specification can import a macro library and use ordinary Lisp code at expansion time. Such macros are useful as genuine shorthands rather than mere decoration: they allow families of related actions, invariants, or structural patterns to be written once and reused declaratively across a model.

Beyond programmable macros, the current Lisp front end now includes several built-in surface shorthands aimed at reducing repetitive boilerplate in larger models. One may write `(vars* _ (p c1 c2) * (market_status mission_state))` to generate a Cartesian product of variable names such as `p_market_status` and `c2_mission_state`. Likewise, `(domain* *_mission_state OPEN RELEASED ...)` applies one finite domain declaration to every declared variable matching a glob pattern. For action predicates, `(assign value to x)` is accepted as readable sugar for `(= x+ value)`, `(equals x y)` as sugar for equality, `(unchanged* pattern ...)` as a glob-based frame-condition generator, and `(alternate-scenarios ...)` as a domain-oriented synonym for `or`. An `if ... then ...` form is also provided; it denotes a conjunction of guard and consequence, and when either branch is written as a list of sibling expressions that list is interpreted as an implicit `and`. These additions keep the language close to the underlying JSON semantics while making large families of similarly shaped state variables substantially easier to read and maintain.

The Tcl-shaped front end, `tdstr`, pursues the same semantic goal through a different host-language discipline. Its basic unit is the Tcl word and command rather than the Lisp list, but the language remains markedly tree-shaped because brace-delimited words preserve internal structure without eager evaluation. A minimal specification has the form:

```
system light-switch {
    vars light
    domain light off on
    init {= light off}
    action turn-on {= light off} {= light+ on}
    action turn-off {= light on} {= light+ off}
    next {or @turn-on @turn-off}
    invariant type-ok {in light {set off on}}
    property eventually-on {eventually {= light on}}
}
```

This notation is not merely a transliteration of the Lisp syntax. It makes use of Tcl's own strengths: braces naturally delimit expressions, commands admit ordinary helper procedures, and command substitution allows list-valued helpers to be spliced into surrounding clauses. A form such as `{*}[unchanged`

`p1 p2 mem]` therefore expands into sibling frame conditions without requiring a separate macro system. The `tdstr` front end also provides built-in shorthands parallel to those of the Lisp DSL: `vars*` for Cartesian variable products, `domain*` for glob-directed domain declarations, `assign`, `equals`, `unchanged*`, `if ... then ...`, and `alternate-scenarios`. In this respect the two surface languages are not competitors so much as alternate presentations of the same modeling idiom.

Two small conventions carry much of the language. First, an unadorned variable name such as `light` or `lock` denotes that variable's value in the current state. A name with the suffix +, such as `light+` or `lock+`, denotes its value in a successor state. Second, an `action` clause is not a command body in the ordinary programming sense. It is a predicate relating current-state values and next-state values. One does not instruct the machine to execute steps imperatively. One characterizes which pairs of states count as valid instances of the named transition. In the `next` clause, a form such as `@turn-on` refers to the named action predicate `turn-on`; larger `next` formulas can combine such references to define the overall transition relation.

This matters more than syntax aesthetics. In a language for behavioral models, verbosity is not a cosmetic flaw. It actively suppresses exploration. If every small change requires plumbing through a large ceremonial representation, users stop trying variants, stop expressing auxiliary properties, and stop thinking in terms of the full state space. A compact DSL preserves the willingness to model.

In `dstr`, the high-level interface is not a veneer over an opaque core. Surface forms are normalized into a clear intermediate representation and then emitted as JSON when needed. This preserves a concise authoring language while retaining a stable machine-oriented form for checking, tooling, and regression tests. The separation is important: it allows the front end to remain pleasant without sacrificing semantic clarity or downstream interoperability.

That separation also makes it possible to support more than one pleasant authoring notation without fragmenting the semantics. The motivation for the Tcl-based sibling language is not that Tcl offers a stronger substrate for DSL construction than Common Lisp; on the contrary, Lisp remains the richer and more principled environment for language extension. Rather, Tcl combines a sufficiently homoiconic command structure with a degree of practical ubiquity that is unusually valuable in constrained environments. It is present by default on many Unix-like systems, commonly available across Linux distributions, and often shipped incidentally even on developer workstations through tools such as Git, whose `git-gui` and `gitk` installations typically bring along a Tcl interpreter. For a lightweight modeling front end, this matters. A language that is slightly less elegant as a meta-language but far more likely to be installed can be the more useful vehicle in practice.

There is a small historical irony here. During the long-running debates over extension languages in free software, Tcl was often criticized in comparison with Lisp-derived alternatives, including in famously sharp polemics from the GNU world. Yet the subsequent ecology of open-source software distributions has often favored deployment convenience over theoretical purity. Without making that history a central thesis, it is worth noting that Tcl's eventual ubiquity—earned through packaging, tooling, and mundane operational fitness—makes it a pragmatic companion to the more language-rich Lisp front end. The important architectural point is that both front ends normalize to the same semantic core.

The macro layer strengthens this separation rather than weakening it. Authors are free to use abstraction aggressively at the surface level, but those abstractions are compiled away before checking. The checker therefore continues to operate on an explicit normalized representation even when the source model was assembled from reusable local idioms. This makes the language feel much smaller to write than the eventual transition relation it denotes.

2.1 Macro Use in Practice

A representative use of the macro system is to load a small library of helper macros, define expansion-time helper functions, and then generate whole families of action clauses rather than writing out near-duplicates by hand. The `dstr` distribution includes examples of this style in its test suite. The following excerpt is representative:

```
(load "cas-macro-lib.lisp")

(defun initial-to-try (pc)
  `(and
    (= ,pc a0)
    (= ,(next-var-symbol pc) 'try)))

(defmacro start-attempt (pc other-pc)
  (cons (initial-to-try pc)
        (preserve-clauses other-pc 'mem)))

(defmacro p1-actions ()
  '((action p1-start
     (!start-attempt p1pc p2pc))
    (action p1-cas-success
      (= p1pc try)
      (= mem 0)
      (= p1pc+ won)
      (!preserve p2pc)
      (= mem+ 1))))
```

Several distinct macro mechanisms appear here. The helper macro `preserve` expands into a list of frame-condition clauses. The form `(!preserve p2pc)` therefore splices `(= p2pc+ p2pc)` directly into an action body. The local macro `start-attempt` combines a helper function with library code to produce a reusable transition pattern, and `(!p1-actions)` or `(!p2-actions)` can then splice whole groups of generated `action` clauses into the surrounding `system`. This is not cosmetic syntax sugar. It is a way to factor out repeated state-update structure while still compiling to the same explicit normalized transition predicates used by the checker.

3 A Language for State and Action

The conceptual center of `dstr` is straightforward: a system is described by variables ranging over finite domains, a predicate selecting the initial states, a set of named actions, and a transition relation assembled from those actions. Around that core sit invariants and reachability-oriented properties.

This point is worth stating explicitly. In `dstr`, an action is written as a predicate over two states: the current state and a candidate successor state. Conditions involving plain variable names constrain the source state; conditions involving the corresponding +-suffixed names constrain the target state. The `next` clause then specifies which action predicates, or which combinations of them, are admitted as actual transitions of the system. Model checking proceeds by finding state pairs that satisfy those predicates, starting from states that satisfy `init`, and assembling the reachable transition graph from there. This organization follows the familiar state-machine style exemplified in TLA+, where one separates an initial-state predicate from next-state actions over current and successor variables [1].

Equally important is the order in which the semantics should be understood. First, the state universe is determined as the Cartesian product of the declared variable domains. Second, each action denotes a predicate over a pair of states: a current state and a candidate successor state. Third, the `next` clause combines these action predicates into a single transition relation on the state universe. Finally, reachability from `init` determines the actual state-transition graph of the system. This way of stating the semantics is clarifying because it separates the space of possible states, the logical definition of allowed transitions, and the reachable behavior that is ultimately explored.

This style of description has an important philosophical consequence. It asks the author to define behavior intensionally rather than procedurally. One says what constitutes a legal state and what changes are allowed, not which control path a single run happened to follow. That shift in viewpoint is exactly what makes enumeration meaningful. Once a system is expressed as a finite state and action

theory, the reachable space can be explored completely.

Explicit enumeration is sometimes dismissed as unsophisticated because it does not aspire to hide the state space behind symbolic compression. For the class of systems targeted by `dstr`, that criticism misses the point. In early design work and in many finite operational models, explicitness is an advantage. It gives exact reachable states, exact transitions, exact deadlocks, and exact counterexample paths. Most importantly, it yields a graph that can be inspected and repurposed without semantic ambiguity.

4 Examples

Examples clarify the intended use of `dstr` more effectively than abstract description alone. The following cases are not included merely as syntax samples. They illustrate the form of feedback produced by the model checker and the way in which that feedback supports design understanding. In each example, occurrences of a variable such as `x` refer to the current state, while `x+` refers to the successor state being constrained by the action predicate.

4.1 A Minimal Switching System

The following specification describes a two-state switch:

```
(system light-switch
  (vars light)
  (domain light off on)
  (init (= light off))
  (action turn-on
    (= light off)
    (= light+ on))
  (action turn-off
    (= light on)
    (= light+ off))
  (next (or @turn-on @turn-off))
  (invariant type-ok
    (in light (set off on)))
  (property eventually-on
    (eventually (= light on))))
```

This is a tiny system, but it demonstrates the basic rhythm of the language. There is one variable, two actions, a simple type discipline expressed as an invariant, and a reachability property. When model checked, the current implementation reports:

Spec: light-switch
Reachable states: 2
Transitions explored: 2
Properties: {eventually-on=true}

Even here the report is useful. It confirms that the intended state space is exactly the one the author had in mind, that the transition relation is neither over-constrained nor under-constrained, and that the desired state is indeed reachable. The resulting graph is small enough to be grasped as a whole, which makes it a useful introductory example both for teaching and for validating the surface notation itself.

4.2 A Product-Structured Market Model

The newer surface shorthands become more useful when several related financial objects share the same fields and transition story. The following excerpt models one master instrument record together

with two venue-specific listings. Each object carries a lifecycle state, a trading-status state, and a reference data version. The actions shown here express a stylized but recognizably financial workflow: enrichment completes, continuous trading starts, trading is paused, trading resumes, and the instrument can ultimately be disabled.

```
(system instrument-lifecycle
  (vars* _ (master venue1 venue2) * (lifecycle trading_status refdata_version))
  (domain* *_lifecycle DRAFT ENRICHING READY ACTIVE HALTED DISABLED)
  (domain* *_trading_status NOT_TRADING TRADING PAUSED DISABLED)
  (domain* *_refdata_version 1 2)
  (init
    (equals master_lifecycle ENRICHING)
    (equals venue1_lifecycle ENRICHING)
    (equals venue2_lifecycle ENRICHING)
    (equals master_trading_status NOT_TRADING)
    (equals venue1_trading_status NOT_TRADING)
    (equals venue2_trading_status NOT_TRADING)
    (equals master_refdata_version 1)
    (equals venue1_refdata_version 1)
    (equals venue2_refdata_version 1))
  (action enrichment-complete
    (unchanged* *_refdata_version)
    (alternate-scenarios
      (if
        ((equals master_lifecycle ENRICHING)
         (equals venue1_lifecycle ENRICHING)
         (equals venue2_lifecycle ENRICHING))
        then
        ((assign READY to master_lifecycle)
         (assign READY to venue1_lifecycle)
         (assign READY to venue2_lifecycle)
         (assign NOT_TRADING to master_trading_status)
         (assign NOT_TRADING to venue1_trading_status)
         (assign NOT_TRADING to venue2_trading_status)))))
  (action start-trading
    (unchanged* *_refdata_version)
    (if
      ((equals master_lifecycle READY)
       (equals venue1_lifecycle READY)
       (equals venue2_lifecycle READY))
      then
      ((assign ACTIVE to master_lifecycle)
       (assign ACTIVE to venue1_lifecycle)
       (assign ACTIVE to venue2_lifecycle)
       (assign TRADING to master_trading_status)
       (assign TRADING to venue1_trading_status)
       (assign TRADING to venue2_trading_status))))
  (action pause-trading
    (unchanged* *_refdata_version)
    (if
      ((equals master_lifecycle ACTIVE)
       (equals venue1_lifecycle ACTIVE)
       (equals venue2_lifecycle ACTIVE))
      then
      ((assign HALTED to master_lifecycle)
       (assign HALTED to venue1_lifecycle)
       (assign HALTED to venue2_lifecycle)
       (assign PAUSED to master_trading_status)
```

```
      (assign PAUSED to venue1_trading_status)
      (assign PAUSED to venue2_trading_status))))
  (action resume-trading
    (unchanged* *_refdata_version)
    (if
      ((equals master_lifecycle HALTED)
       (equals venue1_lifecycle HALTED)
       (equals venue2_lifecycle HALTED))
      then
      ((assign ACTIVE to master_lifecycle)
       (assign ACTIVE to venue1_lifecycle)
       (assign ACTIVE to venue2_lifecycle)
       (assign TRADING to master_trading_status)
       (assign TRADING to venue1_trading_status)
       (assign TRADING to venue2_trading_status))))
  (action disable-instrument
    (unchanged* *_refdata_version)
    (if
      ((not (equals master_lifecycle DISABLED))
       (not (equals venue1_lifecycle DISABLED))
       (not (equals venue2_lifecycle DISABLED)))
      then
      ((assign DISABLED to master_lifecycle)
       (assign DISABLED to venue1_lifecycle)
       (assign DISABLED to venue2_lifecycle)
       (assign DISABLED to master_trading_status)
       (assign DISABLED to venue1_trading_status)
       (assign DISABLED to venue2_trading_status))))
  (next
    (alternate-scenarios
      @enrichment-complete
      @start-trading
      @pause-trading
      @resume-trading
      @disable-instrument)))
```

This example is deliberately more repetitive in its underlying semantic shape than the switch model, and therefore better illustrates the value of the newer surface forms. The `vars*` and `domain*` clauses express a common "master plus venues" product structure without hiding the resulting explicit state variables. The action bodies combine frame conditions generated by `unchanged*` with branch-oriented business logic written using `if`, `assign`, and `alternate-scenarios`. The result is still compiled into the same finite explicit-state JSON representation, but the authoring notation is closer to the way life-cycle and market-status rules are often described in exchange, broker, and reference-data discussions. The same family of shorthands is now available in the Tcl-based `tdstr` front end as well, which is useful when one wants this style of concise, product-structured specification in environments where a Lisp runtime is not readily available.

For comparison, the same model can be written in `tdstr` as follows:

```
system instrument-lifecycle {
    vars* _ {master venue1 venue2} * {lifecycle trading_status refdata_version}

    domain* *_lifecycle DRAFT ENRICHING READY ACTIVE HALTED DISABLED
    domain* *_trading_status NOT_TRADING TRADING PAUSED DISABLED
    domain* *_refdata_version 1 2

    init \
        {equals master_lifecycle ENRICHING} {equals venue1_lifecycle ENRICHING} {equals
```

```
        venue2_lifecycle ENRICHING} \
    {equals master_trading_status NOT_TRADING} {equals venue1_trading_status
        NOT_TRADING} {equals venue2_trading_status NOT_TRADING} \
    {equals master_refdata_version 1} {equals venue1_refdata_version 1} {equals
        venue2_refdata_version 1}

action enrichment-complete \
    {unchanged* *_refdata_version} \
    {
        if
        {
            {equals master_lifecycle ENRICHING}
            {equals venue1_lifecycle ENRICHING}
            {equals venue2_lifecycle ENRICHING}
        }
        then
        {
            {assign READY to master_lifecycle}
            {assign READY to venue1_lifecycle}
            {assign READY to venue2_lifecycle}
            {assign NOT_TRADING to master_trading_status}
            {assign NOT_TRADING to venue1_trading_status}
            {assign NOT_TRADING to venue2_trading_status}
        }
    }

action start-trading \
    {unchanged* *_refdata_version} \
    {
        if
        {
            {equals master_lifecycle READY}
            {equals venue1_lifecycle READY}
            {equals venue2_lifecycle READY}
        }
        then
        {
            {assign ACTIVE to master_lifecycle}
            {assign ACTIVE to venue1_lifecycle}
            {assign ACTIVE to venue2_lifecycle}
            {assign TRADING to master_trading_status}
            {assign TRADING to venue1_trading_status}
            {assign TRADING to venue2_trading_status}
        }
    }

action pause-trading \
    {unchanged* *_refdata_version} \
    {
        if
        {
            {equals master_lifecycle ACTIVE}
            {equals venue1_lifecycle ACTIVE}
            {equals venue2_lifecycle ACTIVE}
        }
        then
        {
            {assign HALTED to master_lifecycle}
```

```
            {assign HALTED to venue1_lifecycle}
            {assign HALTED to venue2_lifecycle}
            {assign PAUSED to master_trading_status}
            {assign PAUSED to venue1_trading_status}
            {assign PAUSED to venue2_trading_status}
         }
      }

   action resume-trading \
      {unchanged* *_refdata_version} \
      {
         if
         {
            {equals master_lifecycle HALTED}
            {equals venue1_lifecycle HALTED}
            {equals venue2_lifecycle HALTED}
         }
         then
         {
            {assign ACTIVE to master_lifecycle}
            {assign ACTIVE to venue1_lifecycle}
            {assign ACTIVE to venue2_lifecycle}
            {assign TRADING to master_trading_status}
            {assign TRADING to venue1_trading_status}
            {assign TRADING to venue2_trading_status}
         }
      }

   action disable-instrument \
      {unchanged* *_refdata_version} \
      {
         if
         {
            {not {equals master_lifecycle DISABLED}}
            {not {equals venue1_lifecycle DISABLED}}
            {not {equals venue2_lifecycle DISABLED}}
         }
         then
         {
            {assign DISABLED to master_lifecycle}
            {assign DISABLED to venue1_lifecycle}
            {assign DISABLED to venue2_lifecycle}
            {assign DISABLED to master_trading_status}
            {assign DISABLED to venue1_trading_status}
            {assign DISABLED to venue2_trading_status}
         }
      }

   next {
      alternate-scenarios
      @enrichment-complete
      @start-trading
      @pause-trading
      @resume-trading
      @disable-instrument
   }
}
```

The comparison is instructive. The Lisp version makes structural abstraction and macro-based refactoring especially natural. The Tcl version sacrifices some of that metalinguistic power, but in return it uses a reader and runtime that are often already present. The two syntaxes nevertheless converge on the same authorial conveniences: product-style variable declarations, pattern-directed domain clauses, explicit but readable frame conditions, and transition rules that can be phrased in a style close to ordinary workflow prose.

Just as importantly, both front ends feed the same graph-export tooling. The wrapper scripts `dstr-to-svg` and `tdstr-to-svg` compile a source model when necessary, emit a normalized JSON specification, derive a Graphviz DOT graph, and then render a viewable image such as SVG or PDF. This is not a mere presentation nicety. In practice, the intermediate graph is often the first artifact shown to reviewers who care less about the authoring notation than about reachable operating modes, dead ends, and permitted transitions. It also provides a convenient checkpoint when debugging a model: one can inspect the generated DOT before asking whether the checker's verdict is surprising. For larger models, the wrappers expose truncation options so that graph generation can degrade gracefully rather than failing outright under a very large universe.

Figure 1 illustrates this intermediate output for the reduced `instrument-lifecycle-small` sample. In practice, both front ends compile to the same normalized JSON form, so the value lies less in printing two nearly identical diagrams than in showing one representative graph and then moving to a behaviorally different example.

A good contrasting example is `cas-2proc-race`, a small concurrent model of two processes racing to perform `CAS(mem, 0, id)` on a shared memory cell. Each process begins in state `a0`, moves to `try`, and can attempt the compare-and-swap at most once. If the shared cell still contains `0`, that process moves to `won` and installs its identifier in `mem`; otherwise it moves to `lost`. The model therefore captures the central safety fact that at most one process can win, while also recording the postcondition that the winner must agree with the final memory value. Its reachability properties additionally show that each process can participate in either role depending on the schedule.

4.3 A Resource Protocol

A slightly richer example models two processes competing for a lock. In `dstr`, one writes the actions that acquire and release the lock and then asks for both safety and reachability:

```
(system mutex-2proc
  (vars p1 p2 lock)
  (domain p1 idle cs)
  (domain p2 idle cs)
  (domain lock free p1 p2)
  (init (= p1 idle) (= p2 idle) (= lock free))
  (action p1-enter
    (= p1 idle) (= lock free)
    (= p2 p2+)
    (= p1+ cs) (= lock+ p1))
  (action p1-exit
    (= p1 cs) (= lock p1)
    (= p2 p2+)
    (= p1+ idle) (= lock+ free))
  (action p2-enter
    (= p2 idle) (= lock free)
    (= p1 p1+)
    (= p2+ cs) (= lock+ p2))
  (action p2-exit
    (= p2 cs) (= lock p2)
    (= p1 p1+)
    (= p2+ idle) (= lock+ free))
  (next (or @p1-enter @p1-exit @p2-enter @p2-exit))
```

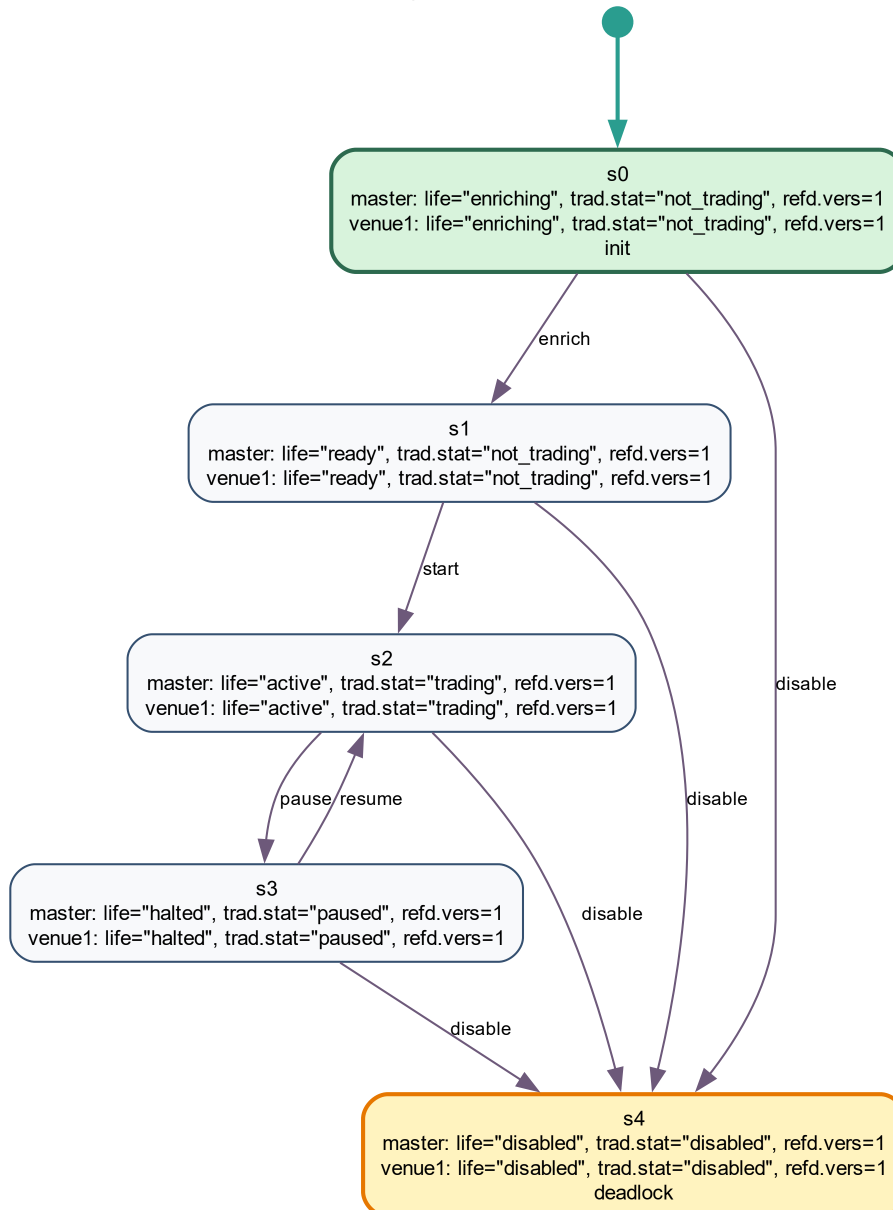

Figure 1: Graph export generated through the `dstr`-side wrapper for the reduced instrument lifecycle model.

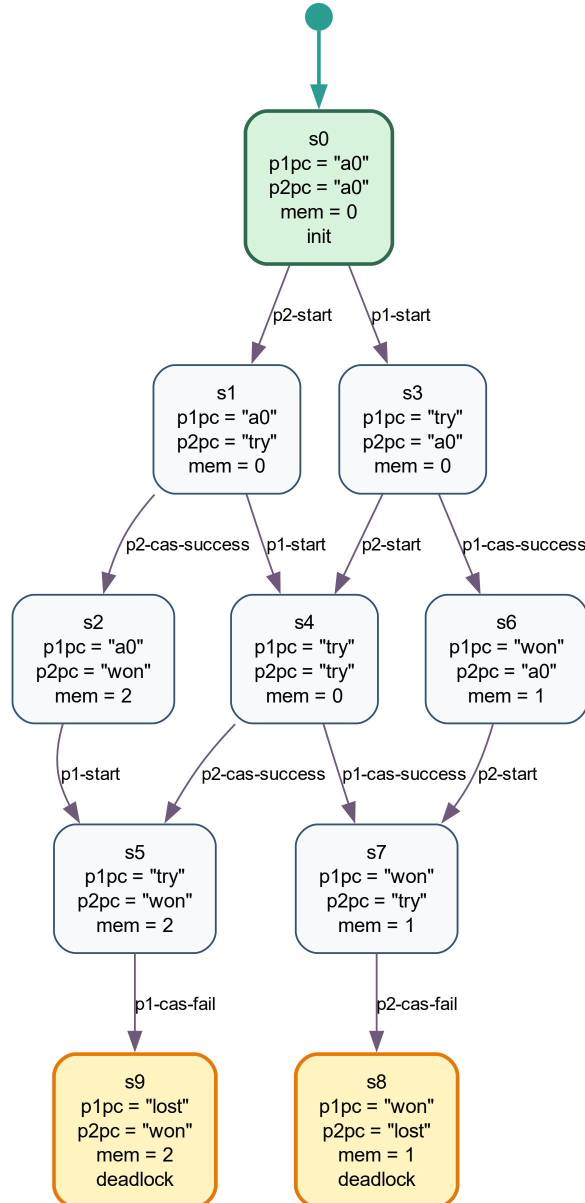

Figure 2: State graph for the `cas-2proc-race` model, showing the competing one-shot compare-and-swap attempts and the winner/loser outcomes they induce.

```
  (invariant mutual-exclusion
    (not (and (= p1 cs) (= p2 cs))))
  (property p1-can-reach-cs (eventually (= p1 cs)))
  (property p2-can-reach-cs (eventually (= p2 cs))))
```

The checker currently reports:

Spec: mutex-2proc
Reachable states: 3
Transitions explored: 4
Properties: {p1-can-reach-cs=true, p2-can-reach-cs=true}

This is the sort of feedback that matters in design. The invariant states that the two processes are never simultaneously in the critical section. The properties establish that each process can still reach that section. Together they distinguish mere exclusion from useful exclusion. The graph then makes the structure behind the report visible: an idle state and two ownership states, with the lock mediating the movement between them. Even in this small example, one can see how a behavioral design becomes a structured object rather than a prose claim.

4.4 A Deliberately Broken Design

The value of a state-action language is equally clear when the design is wrong. Consider a pathological "mutual exclusion" protocol in which one action places both processes into the critical section at once:

```
(system broken-mutex
  (vars p1 p2)
  (domain p1 idle cs)
  (domain p2 idle cs)
  (init (= p1 idle) (= p2 idle))
  (action both-enter
    (= p1+ cs)
    (= p2+ cs))
  (next @both-enter)
  (invariant mutual-exclusion
    (not (and (= p1 cs) (= p2 cs)))))
```

Here the checker does not simply say that the invariant fails. It shows how:

Spec: broken-mutex
Reachable states: 2
Transitions explored: 2
Properties: {}
Invariant violations:
- mutual-exclusion path=[{p1=idle, p2=idle}, {p1=cs, p2=cs}]

That path is exactly the sort of evidence a designer needs. It is short, concrete, and already graph-shaped. One can mark it in a visualization, isolate the error boundary with graph tools, or compare it to a repaired design. This illustrates why explicit-state output is valuable beyond verdicts. It supports diagnosis.

4.5 A Small Search Problem

`dstr` is not limited to protocol-like examples. It also handles finite search spaces naturally. The following model captures a water-jug puzzle with a five-gallon jug and a three-gallon jug:

```
(system water-jugs
  (vars big small)
  (domain big 0 1 2 3 4 5)
  (domain small 0 1 2 3)
  (init (= big 0) (= small 0))
  (action fill-small-jug
    (= small+ 3)
    (= big+ big))
  (action fill-big-jug
    (= big+ 5)
    (= small+ small))
  ...
  (property can-measure-four-gallons
    (eventually (= big 4))))
```

The checked version in the repository includes the expected pouring and emptying actions. The current model checker reports a reachable space of 16 states, explores 74 transitions, and confirms that the four-gallon target is reachable:

> Reachable states: 16
> Transitions explored: 74
> Properties: {can-measure-four-gallons=true}

This example is useful because it is neither purely didactic nor obviously protocol-shaped. It shows that the language can describe constrained transformations over a finite state space and answer a concrete goal question: is the target configuration reachable? Once the graph exists, one can move beyond the yes-or-no answer and ask which subgraph contains all successful strategies, which states are bottlenecks, and how many distinct routes lead to the target.

4.6 A Three-Process Bakery Protocol

A more substantial example is a bounded three-process model of the bakery algorithm. The purpose of including it is twofold. First, it shows that `dstr` can express a recognizably algorithmic coordination protocol in a direct and readable style. Second, it shows what verification means in a case where the model is large enough to be interesting, yet still explicit enough to be fully explored.

The model uses, for each process, a control-state variable, a boolean `choosing` flag, and a ticket number. In the bounded finite model used here, ticket values range from `0` to `4`. The control states are `a0` (idle), `a1` (choosing a ticket), `wait`, and `cs`. The following excerpt shows the overall structure together with the first process's actions:

```
(system bakery-3proc
  (vars p1pc p2pc p3pc ch1 ch2 ch3 n1 n2 n3)
  (domain p1pc a0 a1 wait cs)
  (domain p2pc a0 a1 wait cs)
  (domain p3pc a0 a1 wait cs)
  (domain ch1 t nil)
  (domain ch2 t nil)
  (domain ch3 t nil)
  (domain n1 0 1 2 3 4)
  (domain n2 0 1 2 3 4)
  (domain n3 0 1 2 3 4)
  (init
    (= p1pc a0) (= p2pc a0) (= p3pc a0)
    (= ch1 nil) (= ch2 nil) (= ch3 nil)
    (= n1 0) (= n2 0) (= n3 0))
```

```
(action p1-start
  (= p1pc a0)
  (= p1pc+ a1)
  (= ch1+ t)
  (= p2pc+ p2pc) (= p3pc+ p3pc)
  (= ch2+ ch2) (= ch3+ ch3)
  (= n1+ n1) (= n2+ n2) (= n3+ n3))

(action p1-choose
  (= p1pc a1)
  (= p1pc+ wait)
  (= ch1+ nil)
  (or
    (and (>= n2 n3) (< n2 4) (= n1+ (+ n2 1)))
    (and (> n3 n2) (< n3 4) (= n1+ (+ n3 1))))
  (= p2pc+ p2pc) (= p3pc+ p3pc)
  (= ch2+ ch2) (= ch3+ ch3)
  (= n2+ n2) (= n3+ n3))

(action p1-enter
  (= p1pc wait)
  (not ch2)
  (not ch3)
  (or (= n2 0) (< n1 n2) (= n1 n2))
  (or (= n3 0) (< n1 n3) (= n1 n3))
  (= p1pc+ cs)
  (= p2pc+ p2pc) (= p3pc+ p3pc)
  (= ch1+ ch1) (= ch2+ ch2) (= ch3+ ch3)
  (= n1+ n1) (= n2+ n2) (= n3+ n3))

(action p1-exit
  (= p1pc cs)
  (= p1pc+ a0)
  (= n1+ 0)
  (= ch1+ nil)
  ...)

(next
  (or @p1-start @p1-choose @p1-enter @p1-exit
      @p2-start @p2-choose @p2-enter @p2-exit
      @p3-start @p3-choose @p3-enter @p3-exit))

(invariant mutual-exclusion
  (not
    (or (and (= p1pc cs) (= p2pc cs))
        (and (= p1pc cs) (= p3pc cs))
        (and (= p2pc cs) (= p3pc cs)))))

(property p1-can-reach-cs (eventually (= p1pc cs)))
(property p2-can-reach-cs (eventually (= p2pc cs)))
(property p3-can-reach-cs (eventually (= p3pc cs))))
```

Several features of the language are visible here. The actions are not written as assignment blocks but as predicates over current and successor states. Unchanged variables are stated explicitly, which makes each action a complete binary-state condition. The `p1-choose` action computes a bounded version of “one plus the maximum competing ticket”, while `p1-enter` encodes the waiting condition that the other processes are not choosing and do not hold a lexicographically prior ticket.

The checker explores this bounded model successfully. It reports:

Spec: bakery-3proc
Reachable states: 200
Transitions explored: 390
Properties: {p1-can-reach-cs=true, p2-can-reach-cs=true, p3-can-reach-cs=true}

Most importantly, the mutual-exclusion invariant is preserved throughout the reachable state space. The model also includes two simple structural invariants: that an idle process has ticket `0`, and that a process can have its `choosing` flag raised only while in the ticket-selection control state. These invariants likewise hold over all reachable states.

There is, however, an important modeling caveat. Because the ticket domain is bounded artificially to keep the model finite, the checker also finds deadlocks in states where a process would need a ticket value beyond the chosen bound. This does not contradict the safety result. It reflects the fact that the model is a finite approximation of the unbounded bakery protocol. In this setting, `dstr` is verifying a meaningful safety claim under a deliberate state space restriction, while simultaneously exposing where that restriction begins to matter operationally.

4.7 What the Reports Mean

Across these examples, the current checker reports four kinds of information of lasting value:

- the number of reachable states, which tells us the effective behavioral size of the system,
- the number of explored transitions, which gives a first sense of the system's dynamical richness,
- property results, which answer the author's stated reachability questions, and
- invariant violations with concrete paths, which turn a failed design into an inspectable artifact.

The importance of these reports lies not only in certifying or refuting claims. They also prepare the model for its second life as a graph. Counts orient the reader, property results identify semantically interesting regions, and counterexample paths provide anchors for subsequent graph slicing and analysis.

5 Why the Graph Matters

The state graph produced from a `dstr` model is the central artifact of the method. It records not only what is possible, but how possibility is organized. Which regions are tightly connected? Which actions open or close whole parts of the graph? Where do safe and unsafe states lie relative to one another? Which subgraphs correspond to progress, stuttering, obstruction, or recovery?

A graph answers such questions in a form richer than a pass/fail verdict. Checking tells us whether a property holds, but the graph reveals why the system has the shape it does. This distinction is practical rather than rhetorical. Once the graph exists, it can support explanation, debugging, design comparison, teaching, and publication.

For that reason, `dstr` is best understood not simply as a specification language or a checker, but as a graph generator for behavioral systems.

6 Graph Query as a Second Stage

This perspective becomes much more powerful when paired with a graph query language. The role of `greggle` in this ecosystem is precisely that. If `dstr` produces a labelled directed graph of reachable behavior, then `greggle` provides a way to ask structured questions about paths, patterns, and relational

configurations inside that graph. The companion tools greggle and gruggle are described in more detail in earlier work on regular path queries and graph manipulation [2].

This pairing is compelling because the two layers address different questions. dstr is about *stating* a dynamic system. greggle is about *interrogating* the graph that emerges. One can imagine asking:

- whether there exists a path from an initial state to a region marked by a particular action pattern,
- whether unsafe states are reachable only through some narrow corridor of intermediate states,
- whether two families of states are connected by alternating classes of transitions, or
- whether there are recurrent motifs in the structure of recovery or escalation.

These are graph questions rather than front-end specification questions. Treating the reachable system as a first-class graph allows analysis to proceed at the right level of abstraction. The state-action language establishes the world; the graph query language studies its internal geometry.

7 Graph Manipulation as Presentation and Analysis

After querying comes shaping. Large graphs are rarely useful in raw form. They need to be sliced, filtered, annotated, highlighted, and projected for a human reader. This is where gruggle fits naturally beside dstr and greggle.

gruggle treats a graph as an object that can be manipulated without losing its identity as a graph. Paths can be retained or discarded. Nodes can be selected by structural criteria. Styles can be attached to the pieces that carry the explanatory burden. A dense state space can therefore be turned into a sequence of views: the safe core, the error boundary, the progress skeleton, the interesting corridor, the counterexample trace, the neighborhood of a deadlock.

In this combined workflow, dstr supplies the behavioral universe, greggle finds the meaningful substructure, and gruggle produces a graph shaped for thought and communication. This is more than tool interoperability. It is a coherent methodology:

1. describe a system in a compact state-action language,
2. enumerate its reachable behavior,
3. interrogate the resulting graph with structured path queries, and
4. manipulate the graph into explanatory views.

8 Design Commitments

The ambition of dstr is not to be maximal. It is to make a particular design space tractable and productive. The language is intentionally small, finite-state oriented, and explicit. Those constraints are not admissions of defeat. They are what make the whole pipeline concrete.

Because the models are finite and enumerable, the resulting graphs are not hypothetical semantic objects. They are actual artifacts that can be exported, visualized, queried, versioned, and compared. Because the DSL is succinct, the cost of expressing another system, another invariant, or another variant remains low. Because the intermediate form is simple, the tooling around it can remain transparent.

This is a useful stance for exploratory formal work. Rather than beginning with a grand universal formalism, one begins with an effective language for stating systems that matter, and then insists that the semantics become explicit enough to support downstream graph reasoning.

9 Use Cases

`dstr` is especially well suited to situations where one wants both formal discipline and graph-level understanding:

- coordination protocols whose failure modes are easier to see as paths and regions than as isolated traces,
- controller logic where the state space is finite but the interaction among modes is subtle,
- workflow and policy systems where reachability, blockage, and recoverable states matter,
- teaching settings where students benefit from seeing how behavioral rules compile into a concrete transition graph, and
- research settings where the state graph is not the endpoint but the input to a richer graph-analytic pipeline.

The last case is especially important. Once behavioral models are routinely turned into graphs, new forms of analysis become natural: graph differencing between design revisions, graph querying for structural motifs, graph projection for role-specific views, and graph rewriting for what-if transformations.

10 A Graph-Centered View of Formal Modeling

There is a broader philosophical claim underneath `dstr`. Formal modeling is often imagined as a narrow act of verification: write a formal description, ask whether a theorem holds, and stop there. But a formal model can be more than an instrument for yes-or-no answers. It can be a generative source of structured objects that participate in a larger computational ecosystem.

In `dstr`, the formal model generates a graph. That graph can then be handled by general graph tools, not merely by the checker that produced it. This creates a bridge between behavioral modeling and graph computation. The benefit is conceptual as much as practical. It invites us to regard state spaces as objects of ongoing inquiry rather than hidden machinery inside verification.

11 Conclusion

`dstr` proposes a simple but powerful workflow. Start with a concise s-expression language for describing state, action, and property. Normalize that description into a stable machine form. Enumerate the reachable behavior explicitly. Treat the resulting state graph as a first-class artifact. Then use query and manipulation tools such as `greggle` and `gruggle` to study and present that graph with far greater precision than a checker report alone can offer.

Seen this way, `dstr` is not merely a compact formalism for finite systems. It is an entry point into a larger graph-centered style of reasoning about behavior. That style is attractive because it keeps three virtues in view at once: the expressive clarity of the surface DSL, the semantic firmness of explicit state exploration, and the analytic freedom of downstream graph query and graph manipulation.

References

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

[2] Sambuddha Majumder, Jayanta Majumder, and Partha P. Chakrabarti. Greggle & gruggle: Composable regular path queries and graph manipulation. viXra:2601.0116, January 2026. Submitted 25 January 2026.