TYPE-THEORETIC HABITATION CHECKS IN SYMBOLIC FINITE AUTOMATA UNDER THREE-VALUED SUBTYPING

JIM NEWTON

EPITA Research Laboratory (LRE), F-94270 Le Kremlin-Bicêtre, France
e-mail address: jnewton@lrde.epita.fr

Abstract. In classical finite automata theory over finite alphabets, questions of language inclusion and intersection are routinely addressed by checking vacuity or habitation of the language of the resulting automaton—operations that are both decidable and deterministic. However, when extending the theory to symbolic finite automata over infinite alphabets with semi-Boolean subtype predicates (returning true/false/don't-know), these operations can become undecidable and nondeterministic.

We show that, despite this divergence, the resulting three-valued vacuity/habitation question reduces to a single shortest-path computation on a weighted graph derived from the automaton. The reduction yields one of three verdicts: vacuous, with no accepting computation; inhabited, with a certified witness; or indeterminate, with a minimal candidate witness that is not certified, because every accepting computation traverses at least one transition whose satisfiability could not be decided. In every case a shortest possible witness—or a proof that none exists—is produced, which is a genuine strength of the reduction and not merely a byproduct, since a minimal failing sequence is what a developer actually debugs.

This theorem is not incidental to the rest of the paper: it is what makes the testing methodology we describe well-defined at all, since checking whether a symbolic automata library preserves language equality across a round trip reduces to exactly this vacuity check, applied to the difference automaton of the two results.

We substantiate this connection empirically with three differential test oracles of increasing power, exercised across three independent σCDFA implementations, in Clojure, Scala, and Python. We characterize the blind spots of the weakest oracle—a round-trip equivalence check reported in earlier work—and show that the stronger oracles, which compare each implementation against its own specification and against one another, catch bugs the weakest one misses, including a bug present in only one of the three implementations and invisible to any single-implementation test.

1. Introduction

When extending classical finite automata theory to infinite alphabets, many core intuitions continue to hold. For example, regular expressions still correspond to equivalence classes of finite automata, and vice versa. Nondeterministic finite automata (NFAs) can still be determinized. A regular expression can still be extracted from a finite automaton, and the language of a regular expression can still be understood—at least in most practical cases—by examining an equivalent automaton. Boolean combinations of languages can likewise be represented using either regular expressions or their corresponding automata.

Preprint submitted to
Logical Methods in Computer Science

However, when symbolic automata are introduced—especially over infinite alphabets with semi-Boolean (true/false/don't-know) subtype predicates—new challenges emerge: undecidable operations and non-determinism which are introduced by semi-Boolean predicates have no counterpart in the classical theory.

What makes this line of research compelling is precisely this: while much of symbolic automata theory mirrors the classical case, subtle but important divergences arise. A goal of this paper (in Section 3) is to explain one such divergence. In the classical setting, vacuity and habitation checks are always decidable and deterministic. But under our generalization, these checks can become undecidable—a difference that has serious implications for both theory and practice.

While much of this paper is theoretical, its formulation depends on a careful adaptation of concepts from classical automata theory to the symbolic setting. For this reason, a significant portion of the paper (Section 2) is devoted to revisiting foundational constructs in the context of symbolic finite automata. Although these aspects are well-understood in the classical theory, their symbolic analogs introduce subtle complexities that must be addressed explicitly in order to make our main result both meaningful and verifiable.

1.1. **A Motivating Example.** Consider a Lisp macro that dispatches on an argument list matched against several regular patterns over types rather than a fixed arity—Clojure's `destructuring-case` and its relatives (`dsdefn`, `rte-case`), which choose the first clause whose pattern matches the call [New25a]. For example,

```
(defn classify [& args]
  (destructuring-case args
    [[a b]      {a Long, b Long}]    :two-integers
    [[a [b]]    {a Long, b String}]  :integer-then-string
    [[a & more] {a String}]          :string-first))
```

defines a function `classify` which dispatches on shape and type of its arguments. At macro-expansion time—before any call is made—the macro must decide, about the patterns given by its clauses:

- **Reachability.** Is a pattern's language[1] non-empty? A clause that can never match—one requiring an argument that is at once `(satisfies even?)` and `(satisfies odd?)`, say—is dead code.
- **Non-overlap.** Do two patterns share a matching sequence? If so, a call can match both and dispatch is ambiguous.
- **Equivalence.** Does a hand-simplified pattern denote the same language as the original?

Each reduces to a vacuity check: reachability is habitation of one σCDFA, non-overlap is vacuity of an intersection, and equivalence is vacuity of a symmetric difference (Proposition 4.1).

In a syntactic type system these checks are routine. They stop being routine as soon as the subtype relation itself can be indeterminate, for many possible reasons, some of which we describe in Section 2.6. The subtype procedure must simply be allowed to answer `dont-know` whenever it meets any such case, and every construction that rests on it—determinization, minimization, the difference automaton, the vacuity check itself—inherits a third outcome.

[1]The language of a pattern is the (possibly infinite, possibly empty) set of all sequences that the pattern matches.

This is not merely hypothetical: Figure 8 (Section 4.9) shows a σCDFA pair drawn from our own generated test corpus whose only accepting computation passes through a transition combining `(satisfies positive?)` with a numeric subtype. The checker can neither prove that computation dead nor certify it; it can only fail to find a witness, and must report `dont-know` rather than a false assurance.

Carrying that third outcome faithfully through the theory (Sections 2 and 3), and testing the libraries that implement it [New24d, New24c, New24a, New25b] (Section 4), is the subject of this paper. We maintain three independent implementations of this theory, in Scala [OAC+04, OZ05] and Clojure [Hic08], both of which run on the JVM, and in Python [vRD11], chosen deliberately as a non-JVM implementation. A fourth, earlier implementation in Common Lisp [Ans94, New24b] predates the semi-Boolean extension this paper studies and implements only the classical, two-valued regular-type-expression theory; it is not counted among the three above, but its host language's own `SUBTYPEP` function (Section 3) is the historical origin of the semi-Boolean predicate idea developed here.

1.2. **Our Contribution.**

(1) A reduction of the three-valued vacuity/habitation question for a σCDFA to a single shortest-path search, which also returns a minimal habitation witness (Theorem 3.3, Section 3), which Section 4's testing oracles rely on to be well-defined and to shrink a failing test case to a minimal witness.
(2) Three differential test oracles for σCDFA libraries, a precise account of the blind spots of the round-trip oracle used in earlier work, and the bugs the stronger oracles expose—including one present in only one of three independent implementations (Section 4).
(3) A finite-automata proof of the derivative-of-complement identity $\partial_\nu\, !r = !\partial_\nu\, r$, clarifying a sketch from [New25b] (Section 2.12).
(4) Structure-first generators for random σCDFAs and a statistical characterization of the resulting corpus (Sections 4.8 and 4.9).

Why does this paper combine a shortest-path reduction with a testing methodology, rather than presenting either alone? The dependency is symmetric, not one-directional. A theory-only account of Theorem 3.3 would need to explain why a three-valued vacuity verdict matters in the first place; the answer here is that checking whether a round-trip through a symbolic automata library preserves the language reduces to exactly this vacuity check, on the difference automaton (Proposition 4.1). A testing-only account of Section 4's oracles, conversely, would need to explain why deciding vacuity of the difference automaton is hard or interesting in the first place; the answer is that under semi-Boolean predicates it is neither the classical decidable nor the deterministic question it is for ordinary automata, and making the reduction precise—with a minimal witness—is what makes automated shrinking of a failing test case tractable. Each half motivates the other.

The theorem is not tied to testing alone, either: the dispatch-macro scenario of Section 1.1 needs the same reduction, for the same reason—reachability, non-overlap, and equivalence of RTE patterns all reduce to vacuity checks—with no reference to testing at all. We present the theorem primarily through its testing motivation because that is its actual discovery context, and because testing draws on the theorem's full three-way verdict and minimal witness more essentially than dispatch-clause reachability does, which needs only a binary answer.

1.3. **Previous Work.** Brzozowski [Brz64] introduced the derivative in 1964. Owens *et al.* [ORT09] *modernized* the algorithm and applied to regular pattern recognition for sequences of characters, noting that a practical obstacle to this approach is computation intensity when generating finite automata over excessively large alphabets.

In [New18], Newton used deterministic finite automata (DFA) and RTEs to recognize heterogeneous Common Lisp [Ans94] sequences based on type patterns. Newton and Verna [NV18a] addressed certain meta programming aspects of RTEs. Newton [New25b] generalized RTEs to a wider range of programming languages.

D'Antoni and Veanes [DV17b] and Newton [New25b] argue that symbolic FA with generalization that infinite alphabets be partitioned into finitely many equivalence classes retains many *good properties* of the finite-alphabet counterpart; we make this correspondence precise as Proposition 2.15.

Deciding equivalence for symbolic automata and transducers is itself an established line of work: D'Antoni and Veanes give a difference-based decision procedure for extended symbolic finite transducers [DV13] and an alternative, bisimulation-based one for nondeterministic symbolic finite automata [DV17a]. Proposition 4.1 answers a three-valued version of the same question for σCDFA: under semi-Boolean predicates, the vacuity of the difference automaton that decides equivalence in the classical case is not always itself decidable. Symbolic *containment*, of which emptiness is a special case, is solved directly for extended regular expressions by Keil and Thiemann [KT14], over what Fisman, Frenkel, and Zilles [FFZ20] formalize as an *effective Boolean algebra*—a domain and a set of predicates closed under the Boolean connectives whose satisfiability is required to be decidable. Our semi-Boolean predicates (Section 2.4) have exactly this shape except for that one requirement: an effective Boolean algebra is the special case of a semi-Boolean predicate structure in which `dont-know` never occurs. D'Antoni and Veanes lift classical automaton minimization to the symbolic tree-automaton setting [DV16] under the same decidable-satisfiability assumption; Section 3 is in part an account of what changes once that assumption is dropped.

A different, older kind of transition-relation incompleteness comes from digital-logic design: an *incompletely specified finite-state machine* leaves some transitions as "don't care," free for the designer to resolve however is convenient. Minimizing such a machine is already NP-complete [Pfl73], and the difficulty persists in more recent settings: minimizing a deterministic ω-automaton under an arbitrary set of don't-care words is NP-hard and need not even have a unique minimal automaton [LS23]. The resemblance to our own setting is only partial, and worth stating precisely: a don't-care entry is a *design freedom*—the machine's builder chooses it, and nothing is actually unknown—whereas our semi-Boolean `dont-know` is *epistemic*—the true satisfiability of a predicate is fixed and well-defined, we simply cannot always compute it.

Grigore [Gri16], Kennedy and Pierce [KP07], discuss subtype decidability in Java [GJS$^+$14], Scala [OAC$^+$04, OZ05], C#, and .NET Intermediate Language. The work curiously lacks citations for C# and .NET IL, as if the reader is already intimately familiar with C# and .NET.

In Section 2.3 we discuss types as sets. Newton [NP21] introduced SETS to *designate* types with set semantics in Python [vRD11], Clojure [Hic08], and Scala [OAC$^+$04], inspired by the Common Lisp [Ans94] type system. Frisch [FCB08] *et al.* discuss set-based type semantics in non-dynamic languages, arguing that treating types as sets leads to cardinality contradictions in such systems. We avert this pitfall, because no dynamic language that we use supports run-time membership checks of non-trivial function types: $f \in X \to Y$.

Common Lisp [Ans94, Section TYPEP] explicitly forbids this runtime check—for the same reason `SUBTYPEP` may decline to answer at all (§2.4): deciding $f \in X \to Y$ needs exactly the subtype reasoning `SUBTYPEP` is not always able to complete, so rather than let that indeterminacy surface as a silent or ambiguous answer, `TYPEP` is required to signal an error instead.

Clojure Spec [Mil22, Hic20, Hic08], `metosin` [Hei22], and `seqexp` [gra14] support forms of type pattern recognition. Spec is not based on finite-automata theory at all, but rather on the backtracking NFA (non-deterministic finite automaton) approach of Might *et al.* [MDS11], and thus incurs exponential worst-case complexity. As far as we can tell, `seqexp` does not use a finite automata approach.

2. Symbolic Finite Automata

In this section we develop the theoretical and practical construction of symbolic finite automata (σFA). The automaton-theoretic machinery—determinization, complementation, the correspondence with classical automata—follows familiar patterns from classical finite automata theory, adapted to the fact that transition labels are set-valued rather than single symbols; a reader already fluent in that theory can skim §2.2 and §2.8 for the shape of the definitions without much loss. What has no classical-automata-theory counterpart, however, is the type layer underneath the labels: §2.1, §2.3, §2.4, and §2.6 develop the type system (SETS) that transition labels are drawn from, and establish that basic questions about it—is this type inhabited? is one type a subtype of another?—need not be decidable. This indeterminacy, not the automaton structure itself, is the source of the paper's central technical difficulty, so we encourage even a reader expert in classical automata theory to read those subsections closely.

2.1. What is a Type?

Definition 2.1 (complemented type space, type designator, type)**.** Let Σ be a possibly infinite set and Υ be a recursively defined set of symbols each corresponding to a subset of Σ. For each $\nu \in \Upsilon$, let $[\![\nu]\!]$ denote the subset of Σ corresponding to ν. ν is called a *type designator*, and $[\![\nu]\!]$ is called a type. (Σ, Υ) is called a *complemented type space* if:

(1) **Terminal**: There are symbols $\Sigma, \emptyset \in \Upsilon$, with $[\![\Sigma]\!] = \Sigma$ and $[\![\emptyset]\!] = \emptyset \subseteq \Sigma$.
(2) **Complement**: $\nu \in \Upsilon \implies \exists\, \overline{\nu} \in \Upsilon$ with $[\![\overline{\nu}]\!] = \overline{[\![\nu]\!]} \subseteq \Sigma$.
(3) **Intersection**: $\nu, \mu \in \Upsilon \implies \exists (\nu \cap \mu) \in \Upsilon$ with $[\![\nu \cap \mu]\!] = [\![\nu]\!] \cap [\![\mu]\!] \subseteq \Sigma$.
(4) **Union**: $\nu, \mu \in \Upsilon \implies \exists (\nu \cup \mu) \in \Upsilon$ with $[\![\nu \cup \mu]\!] = [\![\nu]\!] \cup [\![\mu]\!] \subseteq \Sigma$.

As a conventional abuse of notation, Σ represents both the universal set (the universal type) and also the type designator. Likewise, the symbol $\emptyset$ is used both as the empty set as well as the designator thereof. To clarify notation, $\Sigma \in \Upsilon$ and $\emptyset \in \Upsilon$ are type designators, while $\Sigma \subseteq \Sigma$ and $\emptyset \subseteq \Sigma$ are types.

The definition above is agnostic to how Υ is actually represented in an implementation, and our own three do not agree: each uses whatever representation is idiomatic for building an AST-like structure in its host language, not a single shared design. In Clojure, a type designator is a plain, homoiconic s-expression—a class or symbol (`Long`), a keyword for the two terminal types (`:sigma`, `:empty-set`), or a tagged list (`(and A B)`, `(satisfies f)`, `(member a b c)`)—and every operation (`typep`, `canonicalize-type`, `subtype?`, `inhabited?`) is a

multimethod dispatching on the list's head symbol; there is no dedicated data type at all. This is not simply the path of least resistance in a Lisp: it is closely modeled on Common Lisp's own built-in *type specifiers* [Ans94, Section 4.2.3], which use exactly this shape—atomic type specifiers are symbols, compound ones are lists whose head names the specifier, drawn from a fixed vocabulary that includes `and`, `eql`, `member`, `not`, `or`, and `satisfies`, the same names `genus.clj` dispatches on. Where Common Lisp's type specifiers are evaluated by the implementation's own built-in type system, ours reimplements the same grammar as ordinary library-level data. In Scala, by contrast, a type designator is a genuine object tree: an abstract class `SimpleTypeD` at the root, with case classes at the leaves (`SAtomic`, `SAnd`, `SOr`, `SNot`, `SMember`, `SEql`, `SSatisfies`, `STop`, `SEmpty`) dispatched through ordinary virtual methods.[2] Python's implementation is a direct, self-acknowledged structural port of the Scala one, down to the same class names, substituting `ABCMeta` for Scala's `sealed`/case-class machinery; its own source says as much, in a docstring reading "this class is just here to emulate the `TerminalType` trait in Scala."

That three implementations disagree about how to represent a type designator at all, yet compute identical subtype, inhabitation, and canonicalization results throughout, is itself evidence for the host-language-agnosticism this paper claims (§2.3): the algebra, not any particular representation, is what travels.

The s-expression representation, however, comes with a genuine cost an algebraic data type does not share: it is difficult to confirm that a given value is actually a well-formed type designator, whether supplied directly by a user as already invalid, or constructed, incorrectly, by a buggy simplification rule. In Scala, `SAnd(1, 2)` is a compile error, since `SAnd`'s constructor requires `SimpleTypeD` arguments; in Clojure, `(and 1 2)` is, structurally, just as valid a piece of data as `(and integer string)`—nothing about the representation itself distinguishes a well-formed designator from arbitrary data shaped like one. Our Clojure implementation answers this with a separate `valid-type?` multimethod, invoked both at the public type constructors and at the entry to canonicalization, that recursively checks exactly this, and it is enough to catch `(and 1 2)`. But `valid-type?`'s own documentation is explicit that it checks only *syntactic* correctness: no validity check, of a syntax tree or of an algebraic data type alike, can catch a simplification rule that produces a syntactically well-formed designator which simply denotes the wrong type. That residual gap is a genuine reason for the differential-testing methodology of Section 4, independent of any argument for it given there: a validator is necessary but not sufficient, and testing against independent oracles is what the gap actually needs.

The compensating advantage is homoiconicity: what you see is what you get. A type designator can be printed, read back, pattern-matched, and transformed with the host language's own ordinary data-manipulation tools, with no separate parser or pretty-printer to keep in sync with the representation—the predicate-rewriting heuristic of §2.5 is itself an example, inspecting a predicate's own source and constructing its replacement directly as data, through no dedicated API.

---

[2]Deliberately *not* `sealed`, even though a closed hierarchy is exactly what this looks like it wants: Scala's `sealed` traits cannot span multiple files, and this package is one file per case class by design. A workaround exists—an unsealed base class carrying shared implementation, with a separate `sealed` trait mixed in purely for exhaustiveness checking—but is itself considered unconventional; see `https://contributors.scala-lang.org/t/possibility-to-spread-sealed-trait-to-different-files/5304` and `https://users.scala-lang.org/t/refactoring-class-hierarchy-into-adt/6997`.

Definition 2.2 (partition)**.** A *partition* of set S is a set of mutually disjoint, possibly empty subsets of S whose union is S.

Definition 2.3 (disjoint symbolic decomposition)**.** Let (Σ, Υ) be a complemented type space. If $S = \{\mu_1, \mu_2, \ldots, \mu_m\} \subseteq \Upsilon$, then the set $D = \{\nu_1, \nu_2, \ldots, \nu_n\} \subseteq \Upsilon$ is called a *disjoint symbolic decomposition* of S if

(1) $1 \leq i \leq n$, $1 \leq j \leq m \implies (\llbracket \nu_i \rrbracket \subseteq \llbracket \mu_j \rrbracket) \vee \left(\llbracket \nu_i \rrbracket \subseteq \overline{\llbracket \mu_j \rrbracket}\right)$.
(2) $\{\llbracket \nu_1 \rrbracket, \llbracket \nu_2 \rrbracket, \ldots \llbracket \nu_n \rrbracket\}$ is a partition of S.

We usually speak of a disjoint symbolic decomposition of Σ rather than of some subset $S \subseteq \Sigma$.

Definition 2.4 (default decomposition, standard partition)**.** Let (Σ, Υ) be a complemented type space. Given distinct $\nu, \mu, \nu_1, \nu_2, \ldots, \nu_n \in \Upsilon$, the *default decomposition* $DD : 2^\Upsilon \to 2^\Upsilon$ and the *standard partition*, $SP : 2^\Upsilon \to 2^{2^\Sigma}$, are defined as

$$DD(\nu, \mu) = \left\{\nu \cap \mu, \overline{\nu} \cap \mu, \nu \cap \overline{\mu}, \overline{\nu} \cap \overline{\mu}\right\} \qquad \subseteq \Upsilon \qquad (2.1)$$

$$DD(\nu_1, \nu_2, \ldots, \nu_n) = \bigcup_{\nu \in DD(\nu_2, \ldots, \nu_n)} DD(\nu_1, \nu) \cup DD\left(\overline{\nu_1}, \nu\right) \qquad \subseteq \Upsilon \qquad (2.2)$$

$$SP(\nu_1, \nu_2, \ldots, \nu_n) = \{\llbracket \nu \rrbracket | \nu \in DD(\nu_1, \nu_2, \ldots, \nu_n)\} \qquad \subseteq 2^\Sigma \qquad (2.3)$$

The default decomposition (2.1) contains four distinct symbols. However, depending on the relation of the corresponding sets, $\llbracket \nu \rrbracket$ and $\llbracket \mu \rrbracket$, the corresponding partition, SP, may contain $\emptyset$ or may contain less than four sets in some cases as some of the set intersections may fail to be unique; for example if $\llbracket \nu \rrbracket = \llbracket \mu \rrbracket$, $\llbracket \nu \rrbracket \subseteq \llbracket \mu \rrbracket$, or $\llbracket \nu \rrbracket \subseteq \overline{\llbracket \mu \rrbracket}$.

2.2. σFA and σCDFA.

Definition 2.5 (symbolic finite automaton, σFA)**.** A *symbolic finite automaton*, σFA, is a tuple: $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ where:

(1) (Σ, Υ) is a complemented type space.
(2) Q is a finite set of so-called *states*.
(3) $q_0 \in Q$ is the so-called *initial* state.
(4) $F \subseteq Q$ is the set of so-called *accepting* states.
(5) $T \subseteq Q \times \Upsilon \times Q$ is a finite set of so-called *transitions*.

A transition, $(q, \nu, r) \in T$, is also denoted $\textcircled{q} \xrightarrow{\nu} \textcircled{r}$. We refer to $q \in Q$ as the *origin* of the transition, $r \in Q$ as the *target*, and $\nu \in \Upsilon$ as the *label*.

Often, in the literature, a non-deterministic finite automaton is defined as having a *set* of initial states. We have no need for this generalization, as the main purpose of Definition 2.5 is to later define σCDFA in Definition 2.8.

Definition 2.6 (un/satisfiable transition)**.** A transition $\textcircled{q} \xrightarrow{\nu} \textcircled{r}$ is called *unsatisfiable* if $\llbracket \nu \rrbracket = \emptyset$—which is possible even when the designator ν is not the symbol $\emptyset \in \Upsilon$—and *satisfiable* otherwise.

Note that it is possible that $\llbracket \nu \rrbracket = \emptyset \subseteq \Sigma$ without $\nu = \emptyset \in \Upsilon$; *e.g.*, the designator $\nu \cap \overline{\nu} \in \Upsilon$ is syntactically distinct from the designator $\emptyset \in \Upsilon$, yet $\llbracket \nu \cap \overline{\nu} \rrbracket = \llbracket \nu \rrbracket \cap \llbracket \overline{\nu} \rrbracket = \emptyset \subseteq \Sigma$.

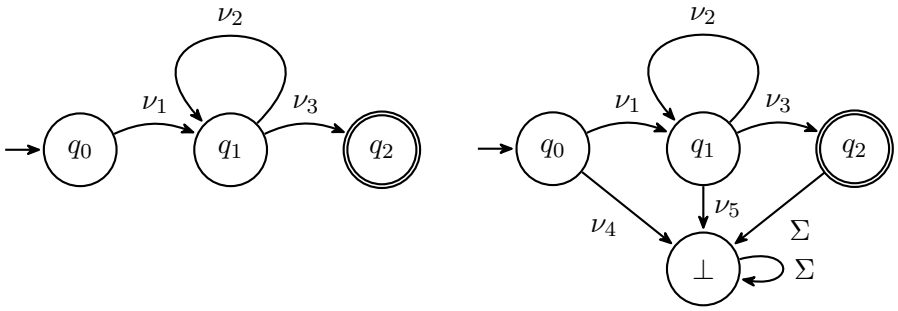


FIGURE 1. Symbolic Finite Automata σFA: (left) incomplete, (right) complete assuming $\nu_4 = \Sigma \setminus \nu_1$, and $\nu_5 = \Sigma \setminus (\nu_2 \cup \nu_3)$. The additional state, $\perp$, called the sink state, is explained in Proposition 2.10.

Definition 2.7 (ξ_A, exiting labels)**.** Let $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ be a symbolic finite automaton. Denote the power set of Υ by 2^Υ. Define the *exiting labels* function, $\xi_A : Q \to 2^\Upsilon$, by $\xi_A(q) = \left\{ \nu \in \Upsilon \,\middle|\, \exists r \in Q,\ \textcircled{q} \xrightarrow{\nu} \textcircled{r} \in T \right\}$.

Definition 2.8 (complete, deterministic, σDFA, σCDFA)**.** A σFA, $A = (\Sigma, \Upsilon, Q, q_0, F, T)$, is said to be

(1) *complete*, if the elements of $\xi_A(q)$ cover Σ; *i.e.*, $\forall q \in Q,\ \bigcup_{\nu \in \xi_A(q)} [\![\nu]\!] = \Sigma$.
(2) *deterministic*, σDFA, if the elements of $\xi_A(q)$ are mutually disjoint; *i.e.*, if $\forall q \in Q, \forall \nu_1, \nu_2 \in \xi_A(q), \nu_1 \neq \nu_2 \implies [\![\nu_1 \cap \nu_2]\!] = \emptyset$.

We denote a complete, deterministic σFA as σCDFA.

As examples, the σFA in Figure 1 (left) is nondeterministic if $[\![\nu_2]\!] \cap [\![\nu_3]\!] \neq \emptyset$, it is deterministic if $[\![\nu_2]\!]$ and $[\![\nu_2]\!]$ are disjoint. It is complete if $[\![\nu_1]\!] = \Sigma$ and $[\![\nu_2 \cup \nu_3]\!] = \Sigma$, regardless of whether $[\![\nu_2]\!]$ and $[\![\nu_3]\!]$ are disjoint.

2.3. **SETS.** We take Υ to be an extension of the built-in type system of a host programming language. In [NP21] and [New25b], this is called a simple embedded type system (SETS)—a pun on the fact that we represent types as sets.

In SETS, we take Σ as the set of all the objects representable as values in the programming language. We define Υ_0 as the set of base type designators in the language, representing all the base types such as integers, strings, and floats. Together with the complement, intersection, and union constructors, these form a complemented type space (Definition 2.1) over Σ.

We do not claim that SETS is not the only way to give Υ a concrete semantics, and we chose it deliberately. Because (Σ, Υ) is closed under complement, intersection, and union by construction (§2.1), arbitrary Boolean combinations of hosted types, singleton types, and predicates come for free—no separate closure step is needed to support the semi-Boolean transition labels used throughout the rest of the paper. Restricting Σ to first-order host values, and never asking whether one function type is a subtype of another, sidesteps the cardinality contradictions Frisch *et al.* identify in set-based semantics for richer type systems that do include function types (§1.3). And, most importantly for what follows, SETS makes the indeterminacy developed in §2.6 an honest reflection of the underlying question rather than an artifact of the encoding: a type designator denotes an actual, well-defined subset of

Σ whether or not we can compute enough about it to decide a subtype or habitation query, so `dont-know` means *undecided*, not *unsupported*. The same SETS algebra is, moreover, not tied to any one host language: our three independent implementations, in Clojure, Scala, and Python, share it despite having almost nothing else in common (§4.4).

In addition to complement, intersection, and union types, we also include:

(1) **Hosted types**: $\Upsilon_0 \subseteq \Upsilon$.
(2) **Singleton types**: $\forall a \in \Sigma$, $\{a\} \in \Upsilon$, with $[\![\{a\}]\!] = \{a\}$.
(3) **Predicates**: for any decidable function $f : \Sigma \to \{true, false\}$, implemented in the host language, $Sat(f) \in \Upsilon$ with $[\![Sat(f)]\!] = \{x \in \Sigma \mid f(x) = true\}$.

An important feature of SETS is that the membership query $x \in [\![\nu]\!]$—given a host value x and a designator ν—is always decidable. The subtype question, however, may not be. The two questions serve different purposes: the subtype question is needed while *constructing* a σCDFA—to decide when transition labels are disjoint or redundant—while the membership query is needed while *using* an already-built σCDFA to match a sequence of actual host values against it. That membership is always decidable is what lets matching proceed reliably even when the automaton's own construction had to tolerate indeterminate transitions. This means the subtype predicate is a semi-Boolean predicate which is allowed to answer *true*, *false*, or `dont-know`. When such type designators are used as labels, indeterminate transitions may arise— the satisfiability of which we are unable to decide. This undecidability is the core source of the difficulty in deciding habitation of a σCDFA as is discussed in Section 3.

2.4. **Semi-Boolean Predicates.** This is not an invention of ours: `SUBTYPEP` has always returned exactly this three-way answer, encoded as two Boolean values—a result and a "was this proven" flag. `(subtypep 'integer 'number)` returns `T`, `T` (proven true); `(subtypep 'number 'integer)` returns `NIL`, `T` (proven false); `(subtypep '(satisfies oddp) 'integer)` returns `NIL`, `NIL`—`dont-know`, spelled as a pair of `NIL` s rather than a keyword. Our semi-Boolean predicates generalize a pattern the CLHS specification has required since long before this library existed.

That escape hatch was not always this precisely bounded. X3J13 cleanup issue SUBTYPEP-TOO-VAGUE[3] records a real portability problem, not a hypothetical one: some implementations "interpreted [the original wording] to mean that it is permissible for `SUBTYPEP` to 'give up' ... in ... cases where it actually would be possible to determine the relationship," making subtype relationships unreliable to depend on in portable code. The committee's fix was not to remove the escape hatch but to bound it precisely, to exactly the eight constructors listed above—the same boundary this paper's own `dont-know` answers are held to.

This three-way distinction is easy to lose unless meticulous care is taken. The JVM's own `Class.isAssignableFrom`, which our Scala implementation itself relies on for atomic host types, offers no such distinction at all: it returns a plain, two-valued `Boolean`, with no way to represent "I could not determine this." A caller receiving `false` cannot tell whether that means *provably not a subtype* or merely *failed to prove it is one*—the two collapse into the same answer, by the very shape of the interface. Our own σCDFA library's `subtypep` deliberately does not inherit this: it returns `Option[Boolean]`, so `None` is a

[3] `https://www.lispworks.com/documentation/HyperSpec/Issues/iss335_w.htm`, "Passed, as amended, Jun89."

distinguishable, first-class outcome from `Some(false)`, built on top of a primitive that offers no such distinction natively.

In Scala, that care is largely enforced automatically: since `Option[Boolean]` and `Boolean` are different types, the compiler itself rejects code that uses one where the other is expected. Our other two implementations have no such backstop, and each is exposed to a different, host-specific version of the same risk. Clojure's `genus` library represents the third value as the keyword `:dont-know`, alongside `true` and `false`; but in a Clojure conditional, every value counts as Boolean true except `false` and `nil` themselves, so `:dont-know` counts as Boolean true there too—passing a three-way result directly to `if`, `and`, or `or` silently treats *don't know* as *yes*, with no type error to catch the mistake. Python's implementation represents the same third value as `None`, alongside `True` and `False`; but in a Python conditional, `None` counts as Boolean false, the same as `False` itself, so the identical carelessness in Python collapses *don't know* into *no* instead—the opposite direction from Clojure, for the same underlying reason: a dynamically typed host's own rule for what counts as true in a conditional does not align with the three-way distinction being represented. Three implementations, three different outcomes for the identical mistake: compile-time rejection, silent *true*, and silent *false*.

For example, deciding $Sat(f) \subseteq Sat(g)$ for user-defined predicates f, g is deciding whether $f(x) \Rightarrow g(x)$ for every $x \in \Sigma$—whether one host-language function implies another—which is undecidable in general, so the procedure returns `dont-know`. Other limitations [NP21] in the base language may also lead to `dont-know`.

2.5. **Predicate Recognition and Host Equality.** Not every predicate a programmer reaches for is genuinely opaque, however. Many idiomatic host-language predicates are themselves defined as a Boolean combination of hosted-type membership tests, merely packaged behind a function name. Clojure's own `int?`, for instance, is defined as `(or (instance? Long x) (instance? Integer x) (instance? Short x) (instance? Byte x))`—already a union of four hosted types, not an opaque computation. Our implementation in Clojure exploits this: before treating a predicate as $Sat(f)$, it inspects the predicate's own source (via `clojure.repl/source-fn`) and, if the body matches this shape—an `instance?` test, or an `or` of several, possibly nested one level to catch predicates defined in terms of other predicates—rewrites $Sat(f)$ to the hosted-type designator it was secretly always computing, e.g. `(satisfies int?)` becomes `(or Long Integer Short Byte)` [NZ25a]. We are fortunate that many built-in type-check predicates match this easily predictable shape: Clojure's core library is remarkably stable, and despite fourteen incremental releases since we first used this technique in 2020—1.10.2 through 1.12.6, spanning the 1.10, 1.11, and 1.12 lines—these core predicate definitions have not changed. The result is decidable by construction, not merely hoped to be so. This is necessarily partial: a predicate with no such structure in its own definition, such as a call to an arbitrary user function, remains genuinely opaque and stays in $Sat(f)$. But for the common case of built-in predicates in heavy idiomatic use, it removes them from the source of indeterminacy entirely, rather than merely mitigating its cost.

A distinct, easier-to-miss assumption is also implicit here: that host equality is a congruence for type membership, i.e. that $x = y$ under the host's default equality implies x and y inhabit exactly the same types. This need not hold. In Python, `1 == 1.0`; in Clojure, `(= '(1 2 3) [1 2 3])`, despite `1/1.0` and list/vector denoting disjoint types. Common Lisp's `EQL`, by contrast, never equates objects of disjoint types, and is what Common Lisp's

type-sensitive operations use by default. This matters because type simplification—used throughout to compute disjointness, subtype relations, and minimal automata—relies on host equality to detect and eliminate redundant terms in `member` and `or` type designators. If that equality is too coarse, a member type such as `(member [1 2 (3 4)] [1 2 [3 4]])` can be silently collapsed to the singleton type `(= [1 2 (3 4)])`, discarding a genuinely distinct element; or a disjunct such as `(= [1])` in `(or (= [1]) (member (1) (2) (3)))` can be dropped as apparently redundant, even though the vector `[1]` is not actually a member of a set of lists.

2.6. **Sources of Indeterminacy.** There are several, sometimes interrelated, ways in which indeterminacy may arise. The most obvious, and the one this paper draws on most often as an example, is predicate types; but an indeterminate subtype relation, or an indeterminate habitation question, can arise from a number of different situations, several of which we enumerate here.

- **Predicate types.** If leaf types include a *predicate*—`(satisfies even?)`, `(satisfies prime?)`, any user-supplied test—then deciding whether `(satisfies even?)` is a subtype of `(satisfies (fn [n] (not (odd? n))))` is deciding whether one host-language function implies another, which is undecidable in general.
- **Open type hierarchies.** Indeterminacy can arise with no predicates at all: if X and Y are known subtypes of Z, but nothing guarantees they are Z's *only* subtypes, then `(and Z (not (or X Y)))` may or may not be empty, and no amount of reasoning about X, Y, and Z alone settles which. This is not a purely Lisp-flavored concern: on the JVM, a classpath scan can enumerate every subclass currently loadable, but dynamic class loading means a later-loaded class could still introduce a new common subtype—the same open-world gap, recurring on an unrelated platform for an unrelated reason.
- **Deliberate incompleteness.** The subtype procedure may decline to fully decide a relation that is decidable in principle, to bound its own cost—but the escape hatch is not unbounded. Common Lisp's `SUBTYPEP` [Ans94, SUBTYPEP] may answer `dont-know` only when at least one argument involves `and`, `eql`, `function` (list form), `member`, `not`, `or`, `satisfies`, or `values`; outside that list, an accurate answer is required. Real implementations exercise this latitude differently. SBCL 2.4.6 [New15] proves `(subtypep '(satisfies integerp) 'integer)`, where ECL 26.5.5 [ECL26] answers `dont-know`; conversely, CLISP 2.49.92 [Hai10] proves `(subtypep '(member 1 3 5) '(satisfies oddp))`, where Clozure CL 1.13 [Clo26] answers `dont-know`. Yet SBCL answers `dont-know` on the second example and CLISP on the first: neither implementation is simply more complete than the other.
- **Unrecognized closed value domains.** Some types have a small, complete, statically-known set of inhabitants—a boxed `Boolean` has exactly *true* and *false*, Scala's `Unit` has exactly one, a host `enum` only its declared constants—so a subtype question against such a type is, in principle, answerable by direct enumeration. In Scala, for instance, `SAnd(SAtomic(classOf[Boolean]), SNot(SMember(true, false)))` denotes exactly the empty set—every boolean is either *true* or *false*—decidable by direct enumeration of `Boolean`'s two inhabitants. A Scala `sealed trait` whose leaves are all `case object`s is closed the same way, but more strongly: the compiler, not a runtime scan, guarantees no further leaf can appear—though actually enumerating those leaves still routes through the same subclass discovery as the **Open type hierarchies** case above, and so inherits its

RTE	Language	RTE	Language	RTE	Language
$\emptyset$	$\emptyset$	$f \cdot g$	$[\![f]\!] \cdot [\![g]\!]$	f^*	$[\![f]\!]^*$
ε	$\{()\}$	$f + g$	$[\![f]\!] \cup [\![g]\!]$	$\lfloor \nu \rfloor$	$\{(a) \mid a \in [\![\nu]\!]\}$
Σ	$\{(a) \mid a \in \Sigma\}$	$f \,\&\, g$	$[\![f]\!] \cap [\![g]\!]$	$!f$	$\Sigma^* \setminus [\![f]\!] = \overline{[\![f]\!]}$

FIGURE 2. Semantics of Rational Type Expressions, assuming f and g are RTEs recognizing languages $[\![f]\!]$ and $[\![g]\!]$.

loopholes unless obtained through a macro instead of reflection. But unless the procedure specifically recognizes this structure, it falls back to treating the predicate as opaque, and reports `dont-know` even though a complete answer was available for free.

- **Simplification erasing evidence.** A sound, denotation-preserving rewrite can still destroy a witness a later query depended on. Since 1 satisfies `odd?`, `(member 1)` is a subtype of `(satisfies odd?)`, and a routine simplification rule collapses `(or (satisfies odd?) (member 1))` to `(satisfies odd?)` alone—the same type, denotationally. But the unsimplified designator is provably satisfiable (witness 1), while the simplified one, checked on its own, is `dont-know`: the witness lived only in syntax that a correct, unrelated transformation removed.

These five are not all the same kind of gap. The first two are genuinely undecidable in general: no amount of computation resolves an arbitrary predicate implication or certifies an open hierarchy's completeness. The next two are decidable for the specific case at hand, just not resolved by this particular procedure, for two different reasons: one a deliberate cost bound, the other a missing structural check. The predicate-rewriting heuristic described above, which recognizes idiomatic predicates like `int?` as secretly decidable hosted-type unions rather than opaque tests, is a real, shipped instance of closing exactly this second kind of gap—not the first. The fifth is different again: nothing here was ever undecidable or unresolved—the unsimplified designator was already correctly determined satisfiable—the gap opens only because a sound, denotation-preserving transformation is under no obligation to preserve the syntactic evidence a later query happens to need. Whether a type is satisfiable is, in this sense, a property of the *designator*, not just the *type* it denotes: two designators with $[\![\nu_1]\!] = [\![\nu_2]\!]$ need not receive the same answer.

A related but structurally different kind of trouble arises for function types, which we do not take up here: the question is not whether a subtype relation is hard to *compute*, but whether *subtype* even has a single, uncontested *definition* for function types at all [Cas16, Section 2.2].

2.7. **Rational Type Expressions.** Sections 2.3, 2.4, and 2.6 developed the type layer that underlies our symbolic automata, and showed how subtype and inhabitation questions may yield the third outcome, `dont-know`. We now return to the automata layer built on top of those same type designators. What follows asks a simple question: once transition labels inherit this indeterminacy, how much of the familiar machinery of classical finite automata theory—regular expressions, determinization, complementation, and derivatives—continues to work?

Definition 2.9 (rational type expression, RTE)**.** Let (Σ, Υ) be a complemented type space. A *rational type expression* (RTE) is any expression defined by Figure 2, representing (recognizing) a language on Σ.

We use RTEs to recognize mixed-type sequences in a programming language. We associate the RTE, r, with a σCDFA $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ so that $\mathcal{L}\{A\} = [\![r]\!]$.

2.8. **Automata Constructions.** Sections 2.1 through 2.7 developed the labels from which symbolic automata are built: type designators, their semantics, and the sources of indeterminacy they may carry. We now turn to the automata constructions themselves. The question guiding this subsection is whether the familiar machinery of classical finite automata theory continues to work when transitions are labeled by potentially indeterminate type designators rather than individual symbols. Propositions 2.10 through 2.18 show that completion, determinization, complementation, and the corresponding language-theoretic constructions remain available in this setting.

Proposition 2.10 (sink state)**.** *Any σFA, $A = (\Sigma, \Upsilon, Q, q_0, F, T)$, deterministic or otherwise, can be made complete by adjoining to Q, one so-called* sink state, $\textcircled{\perp}$*, whose only transition is* $\textcircled{\perp} \xrightarrow{\Sigma} \textcircled{\perp}$*, and adjoining one transition to each state, q:* $\textcircled{q} \xrightarrow{\bigcap_{\nu \in \xi_A(q)} \overline{\nu}} \textcircled{\perp}$*. More explicitly: $A' = (\Sigma, \Upsilon, Q', q_0, F, T')$ is complete provided*

$$Q' = Q \cup \{\textcircled{\perp}\}$$
$$T' = T \cup \left\{ \textcircled{q} \xrightarrow{\nu_q} \textcircled{\perp} \,\middle|\, q \in Q \right\} \qquad \text{where } \nu_q = \bigcap_{\nu \in \xi_A(q)} \overline{\nu}$$

Remark 2.11. This construction is unconditional: it adjoins $\textcircled{q} \xrightarrow{\nu_q} \textcircled{\perp}$ to *every* state q, without first checking whether q is already complete ($[\![\nu_q]\!] = \emptyset$ already)—correctly so, since that check is exactly as hard as the emptiness question it would be used to avoid. A consequence is that the construction is not cheaply idempotent: applying it a second time to an already-complete automaton, without separately tracking that fact, recomputes ν_q over a now-larger set of exiting labels and may add a further, genuinely redundant transition whenever that leftover designator is again only unprovably, rather than provably, empty. Our Scala implementation's `complete` function already anticipates exactly this: it attempts a provable-emptiness check before falling back to the unconditional transition above, and its own comment records the residual case plainly—"it may be that unnecessary transitions are added in the case that some state is locally complete, but we cannot determine it to be so" [New24d].

It is expedient notationally to omit the clutter of drawing the sink state and its incoming transitions. If we claim that the σFA in Figure 1 (left) is complete, then we assume the transitions leading to $\textcircled{\perp}$ are implicit.

Definition 2.12 (sequence, Σ^*)**.** If S is a set, then S^n (*i.e.*, the Cartesian product of n copies of S) denotes the set of sequences of length $n \in \mathbb{N}$. S^0 denotes the set containing only the empty sequence; *i.e.*, $S^0 = \{()\}$.

We often refer to Σ^*, which we define as $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$; *i.e.*, Σ^* is the set of all finite, possibly empty, sequences over Σ.

Definition 2.13 (accepting, satisfiable, rejecting computation)**.** Let $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ be a σFA. A sequence $\left(\overbrace{(q_0, \nu_1, q_1)}^{t_1}, \overbrace{(q_1, \nu_2, q_2)}^{t_2}, \ldots, \overbrace{(q_{n-1}, \nu_n, q_n)}^{t_n}\right) \in T^n$ is said to be a *computation* on A. *I.e.*, $(t_1, t_2, \ldots, t_n)$ designates a connected sequence of transitions from q_0 to q_n—the origin of t_i is the target of t_{i-1}.

The computation is called *satisfiable* if all its transitions are satisfiable, and is called *unsatisfiable* otherwise.

The computation (satisfiable or otherwise) is called *accepting* if $q_n \in F$, and *rejecting* if $q_n \notin F$.

If a σFA is deterministic, then we may unambiguously designate a computation by listing the transition labels, as the first transition always has source q_0, and any state has at most one transition labeled by a particular symbol.

Definition 2.14 (acceptance, language)**.** Let $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ be a σFA, and let $s = (s_1, s_2, \ldots, s_n) \in \Sigma^*$. Then A is said to *accept* (or *match*, or *recognize*) s, if there exists an accepting computation $\left(\overbrace{(q_0, \nu_1, q_1)}^{t_1}, \overbrace{(q_1, \nu_2, q_2)}^{t_2}, \ldots, \overbrace{(q_{n-1}, \nu_n, q_n)}^{t_n}\right)$, for which $s_i \in [\![\nu_i]\!], (i = 1 \ldots n)$.

The set, $\mathcal{L}\{A\}$, of all sequences accepted by A is called the *language* of A.

D'Antoni and Veanes [DV17b] argue informally that a σFA over an infinite alphabet corresponds to a classical finite automaton over the finite alphabet of equivalence classes its labels can distinguish.

The point is not merely that such a correspondence exists. It explains why symbolic automata continue to admit many of the same constructions as classical finite automata. Although the alphabet itself may be infinite, a symbolic automaton can distinguish only finitely many classes of symbols. The next proposition makes this correspondence precise, using the disjoint symbolic decomposition of Definition 2.3, and identifies the classical automaton that is implicitly present beneath a symbolic one.

Proposition 2.15 (correspondence with classical automata)**.** *Let $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ be a σFA with labels $S = \bigcup_{q \in Q} \xi_A(q) = \{\mu_1, \ldots, \mu_m\}$, and let $D = \{\nu_1, \ldots, \nu_n\}$ be* any *disjoint symbolic decomposition of S (Definition 2.3)—for instance MDTD's output* [New23, New25b]*, which builds D algebraically from signed intersections of S (Equation* (2.2)*) and satisfies Definition 2.3 for that reason alone,* even when every `subtypep` query involved answers `dont-know`*. For $a \in \Sigma$ let $\rho(a)$ denote the unique $\nu_i \in D$ with $a \in [\![\nu_i]\!]$, extended symbol-wise to $\rho : \Sigma^* \to D^*$, and let $A' = (Q, D, q_0, F, T')$ be the classical finite automaton with*

$$T' = \left\{(q, \nu_i, q') \,\middle|\, (q, \mu, q') \in T,\ [\![\nu_i]\!] \subseteq [\![\mu]\!]\right\} .$$

Then $\mathcal{L}\{A\} = \{s \in \Sigma^* \mid \rho(s) \in \mathcal{L}\{A'\}\}$. Moreover, if $D = DD(\mu_1, \ldots, \mu_m)$ with every designator denoting $\emptyset$ provably *removed—which needs every emptiness query involved to be decided, never `dont-know`—then ρ identifies exactly the equivalence classes of Σ under "agrees on membership in every μ_j" with the elements of D: D is then, up to this bijection, precisely the finite alphabet A's own labels can distinguish, and no smaller classical automaton computes the same correspondence.*

Proof. By induction on sequence length k: for $k = 0$, both A and A' accept the empty sequence from q_0 iff $q_0 \in F$. For the step, a transition $\textcircled{q} \xrightarrow{\mu} \textcircled{q_1} \in T$ is usable on symbol $a \in \Sigma$ iff $a \in [\![\mu]\!]$. Writing $\nu_i = \rho(a)$, Definition 2.3(1) gives $[\![\nu_i]\!] \subseteq [\![\mu]\!]$ or $[\![\nu_i]\!] \subseteq \overline{[\![\mu]\!]}$; since $a \in [\![\nu_i]\!]$, exactly the former holds when $a \in [\![\mu]\!]$. So $\textcircled{q} \xrightarrow{\mu} \textcircled{q_1}$ is usable on a in A exactly when $\textcircled{q} \xrightarrow{\nu_i} \textcircled{q_1} \in T'$ is usable on $\nu_i = \rho(a)$ in A'. Chaining this correspondence over an entire computation and comparing to Definitions 2.13 and 2.5 [acceptance, language] gives $\mathcal{L}\{A\} = \{s \mid \rho(s) \in \mathcal{L}\{A'\}\}$.

For the "moreover" claim: a and a' agree on membership in every $\mu_j \in S$ exactly when they lie in the same sign-combination of the μ_j's, *i.e.* the same nonempty element of $DD(\mu_1, \ldots, \mu_m)$ (Equation (2.2)). So when D is exactly those provably nonempty elements, $\rho(a) = \rho(a')$ iff a and a' agree on S: ρ identifies exactly the classes of that agreement relation with the elements of D. Without provable emptiness, some designator denoting $\emptyset$ may survive into D unrecognized; it names no symbol of Σ, so it costs the bijection nothing beyond an unreachable letter of A'—the main claim above is unaffected. □

It is worth pausing to interpret what Proposition 2.15 does and does not say. The proposition establishes that a symbolic automaton still admits a classical finite-automata interpretation, despite operating over a potentially infinite alphabet. What changes in the presence of semi-Boolean predicates is not the existence of that correspondence but our ability to compute it as precisely as we would in the classical setting.

Unlike the "moreover" clause, the main correspondence needs no decidability hypothesis at all: any D satisfying Definition 2.3 makes the argument go through, and MDTD's own construction satisfies that definition unconditionally, `dont-know` included—exactly the resilience result established for MDTD elsewhere [New23]: "the MDTD algorithm is guaranteed to compute a set of types which are disjoint; however, they may not be provably inhabited." What semi-Boolean predicates take away, then, is not the existence of a classical automaton underneath a σFA—that survives unconditionally—but the ability to compute the *minimal* one: without decidable emptiness, D may carry designators that denote $\emptyset$ without our being able to tell, so A' may have more letters than Σ actually has distinguishable classes, though never fewer.

Proposition 2.15 established that a symbolic automaton continues to admit a classical finite-automata interpretation. The next question is whether the standard constructions of finite automata theory remain available in this setting. Determinization is a particularly important test case, because it underlies several of the constructions that follow. The next proposition shows that symbolic automata retain this property as well.

Proposition 2.16 (determinization)**.** *Any σFA can be determinized.*

More explicitly, given a (perhaps non-deterministic) σFA, $N = (\Sigma, \Upsilon, Q, q_0, F, T)$, there exists a (deterministic) σDFA $D = (\Sigma, \Upsilon, Q', q_0', F', T')$ for which $\mathcal{L}\{N\} = \mathcal{L}\{D\}$.

Proof. Enumerate $Q = \{q_0, q_1, \ldots q_m\}$.

Define the following sets of transitions, $T_0, T_1, T_2, \ldots, T_m$:

$$T_k = \{(q_k, \nu, p) \mid [\![\nu]\!] \in SP(\xi_A(q_k)),\ \exists (q_k, \mu, p) \in T, [\![\nu]\!] \subseteq [\![\mu]\!]\}$$
$$T' = \bigcup_{k=0}^{m} T_k$$

Now let $N' = (\Sigma, \Upsilon, Q, q_0, F, T')$. By construction of SP (Definition 2.4), any two transitions (p, ν, q_1) and (p, μ, q_2) of T' sharing an origin have labels that designate either the same type, $[\![\nu]\!] = [\![\mu]\!]$, or disjoint types, $[\![\nu]\!] \cap [\![\mu]\!] = \emptyset$. Treating each distinct type appearing as a label in T' as a single letter, N' is a classical finite automaton over a finite alphabet, and $\mathcal{L}\{N'\} = \mathcal{L}\{N\}$; it is determinized by classical techniques [HMU06]. □

Definition 2.17 (complement σDFA)**.** Let $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ be a σCDFA. Let $\overline{A}$ denote the *complement* σFA, $\overline{A} = (\Sigma, \Upsilon, Q, q_0, Q \setminus F, T)$.

Proposition 2.18. *If A is a σCDFA, then*

(1) *$\overline{A}$ is complete and deterministic.*
(2) *$\mathcal{L}\{\overline{A}\} = \Sigma^* \setminus \mathcal{L}\{A\}$, which we denote $\overline{\mathcal{L}\{A\}}$.*

Proof. $\overline{A}$ is complete because it has the exact same transitions, T, as A. $\overline{A}$ is deterministic for the same reason. To show $\mathcal{L}\{\overline{A}\} = \overline{\mathcal{L}\{A\}}$, there are two cases:

(1) If A has no accepting computations, then every computation of A ends in a state not in F, thus ends in a state in $Q \setminus F$. *I.e.*, if $\mathcal{L}\{A\} = \emptyset$ then $\mathcal{L}\{\overline{A}\} = \Sigma^*$.
(2) If t is an accepting computation in A, then it ends in a state $q \in F$, thus does not end in a state in $Q \setminus F$; hence t is rejecting in $\overline{A}$. Likewise if t is a rejecting computation in A, then it ends in a state $q \in Q \setminus F$; hence t is accepting in $\overline{A}$. □

Definition 2.19 (right-automaton)**.** Let $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ be a σCDFA, and let $q_1 \in Q$. Then we call $\overrightarrow{A}_{q_1} = (\Sigma, \Upsilon, Q, q_1, F, T)$ the *right-automaton* [BHM+19] of A at q_1.

Proposition 2.20. *Let $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ be a σCDFA. Let $(q_0) \xrightarrow{\nu} (q_1) \in T$, $s_1 \in [\![\nu]\!]$, and $(s_1, s_2, \ldots, s_n) \in \mathcal{L}\{A\}$. Then $(s_2, \ldots, s_n) \in \mathcal{L}\{\overrightarrow{A}_{q_1}\}$.*

Proof. Let $(\nu_1, \nu_2, \ldots, \nu_n)$ be an accepting computation for $(s_1, s_2, \ldots, s_n)$. *I.e.*, $s_1 \in [\![\nu_1]\!]$, $s_2 \in [\![\nu_2]\!]$, …, $s_n \in [\![\nu_n]\!]$. Thus $(s_2, \ldots, s_n)$ is accepted by the computation $(\nu_2, \ldots, \nu_n)$ on $\overrightarrow{A}_{q_1}$. □

2.9. **Brzozowski Derivative Construction of σCDFA.** Sections 2.8 and 2.7 established the objects we wish to relate: symbolic automata and rational type expressions. We now turn to the construction that connects them. The Brzozowski derivative provides a systematic method for building a σCDFA from an RTE, and the resulting construction is used repeatedly throughout the remainder of the paper. The purpose of this subsection is therefore not merely to introduce another automata-theoretic technique, but to explain how the symbolic automata studied later are actually obtained from the expressions that describe their languages.

In this section we present briefly the Brzozowski construction algorithm generalized for σCDFA using semi-Boolean predicates as labels. This construction is explained in more detail in [New25b]. While Brzozowski/Owens [Brz64, ORT09] defined $\partial_a r$, for the letter $a \in \Sigma$, we define $\partial_\nu r$ for type designator $\nu \in \Upsilon$.

Write $h :: s$ for the sequence obtained by prepending the element $h \in \Sigma$ to the sequence $s \in \Sigma^*$.

$$\partial_\nu \emptyset = \partial_\nu \varepsilon = \emptyset \tag{2.4}$$
$$\partial_\nu (r^*) = \partial_\nu r \cdot r^* \tag{2.5}$$
$$\partial_\nu (r + s) = \partial_\nu r \ + \ \partial_\nu s \tag{2.6}$$
$$\partial_\nu (r \,\&\, s) = \partial_\nu r \ \& \ \partial_\nu s \tag{2.7}$$
$$\partial_\nu !r = !\partial_\nu r \tag{2.8}$$

$$\partial_\nu (r{\cdot}s) = \begin{cases} (\partial_\nu r) \cdot s & \text{if } () \notin [\![r]\!] \\ (\partial_\nu r) \cdot s + \partial_\nu s & \text{if } () \in [\![r]\!] \end{cases} \tag{2.9}$$
$$\partial_\nu \lfloor \mu \rfloor = \varepsilon \quad \text{if } [\![\nu]\!] \subseteq [\![\mu]\!] \tag{2.10}$$
$$\partial_\nu \lfloor \mu \rfloor = \emptyset \quad \text{if } [\![\nu]\!] \cap [\![\mu]\!] = \emptyset \tag{2.11}$$
$$\partial_\nu \lfloor \mu \rfloor \quad \text{otherwise, no rule defined} \tag{2.12}$$

FIGURE 3. Brzozowski Derivative Rules. Variables ν and μ range over type designators ($[\![\nu]\!], [\![\mu]\!] \subseteq \Sigma$); r and s range over RTEs. Equation (2.8) does *not* hold for arbitrary ν; it holds when ν is drawn from a partition of Σ adapted to the symbols of r (Proposition 2.23, Section 2.12).

Definition 2.21. Given a type designator $\nu \in \Upsilon$ and an RTE r, a *Brzozowski derivative*, denoted $\partial_\nu r$, is any RTE such that

$$[\![\partial_\nu r]\!] = \{s \in \Sigma^* | \exists h \in [\![\nu]\!] \ h :: s \in [\![r]\!]\} \ .$$

2.10. **Constructing States and Transitions.** Our construction algorithm entails assigning the given RTE, r, to an initial state, called q_0, deriving a set of type designators $\{\nu_1, \ldots, \nu_n\}$ using an algorithm called MDTD [New23, New25b]. MDTD produces a disjoint symbolic decomposition (Definition 2.3) but eliminates symbols which provably designate the empty type, thus sometimes contains several type symbols representing the empty type, but never containing two types representing the same inhabited type.

Remark 2.22. Consequently, the resulting σCDFA may have states with more than one exiting transition labeled by an unsatisfiable designator, indistinguishable to the construction. This never affects correctness: since no symbol satisfies any of them, they affect neither $\mathcal{L}\{A\}$ nor which computations are accepting—only how many candidate transitions a runtime membership check must attempt before finding the one that matches, or exhausting them all.

Next, construct $q_1, \ldots, q_n$ and associate q_i with $\partial_{\nu_i} r$, computed as prescribed in Figure 3. Unify any states whose associated RTEs are decidable equivalent.

It is guaranteed by construction of state q_i that the RTE $\partial_{\nu_i} r$ is an RTE whose language is exactly as in Definition 2.21. Therefore, we construct the following transitions, with common origin q_0: $(q_0, \nu_1, q_1), (q_0, \nu_2, q_2), \ldots, (q_0, \nu_n, q_n)$. Now repeat the process on newly discovered states and their RTEs. The process terminates because an RTE has only finitely many dissimilar derivatives [Brz64], modulo the similarity rules folded into the derivative computation.

2.11. **Computing the Brzozowski Derivative.** Let r be an RTE and $\nu \in \Upsilon$ be a type designator. The RTE, $\partial_\nu r$, is recursively computed as prescribed in Figure 3. In [New25b], we introduced subtype-based rules (2.10), (2.11) and (2.12) which generalize rules from Owens [ORT09].

In a σCDFA, if there is a transition $\textcircled{p} \xrightarrow{\nu} \textcircled{q}$, with $\mathcal{L}\{A\} = [\![r]\!]$, then

$$\mathcal{L}\{\overrightarrow{A}_q\} = [\![\partial_\nu r]\!] \,, \tag{2.13}$$

as the definition of $\mathcal{L}\{\overrightarrow{A}_q\}$ and $[\![\partial_\nu\, r]\!]$ turn out to be identical.

Owens [ORT09] and Keil [KT14] also give similar recursive rules for computing *nullability*, detecting whether $() \in [\![r]\!]$, as well as $1^{st}(r)$ which is set of symbols which appear as first positions in a regular expression, *i.e.* the set of type designators to which (2.10) and (2.11) will be applied.

2.12. **Derivative of Complement.** The derivative rules of Figure 3 are the foundation of the Brzozowski construction described in Section 2.9. Most of those rules follow directly from Definition 2.21. Equations (2.8) and (2.7) are more subtle. Both are sufficiently natural that one might expect them to hold immediately, yet a direct expansion of Definition 2.21 does not establish either identity. The purpose of this subsection is to close that gap: we show that the identities are valid, and explain why the automata-theoretic structure developed earlier in this section supplies what the direct argument does not.

Equation (2.8), $\partial_\nu !r = !\partial_\nu\, r$, is not immediately evident. Directly applying the definitions from above, we see that Equations (2.14) and (2.15) are not identical. The same is true of Equation (2.7), $\partial_\nu\,(r \,\&\, s) = \partial_\nu\, r \,\&\, \partial_\nu\, s$: unwinding it directly from Definition 2.21, with $t \in \Sigma^*$ for the tail sequence, calls for a single $h \in [\![\nu]\!]$ satisfying both $h :: t \in [\![r]\!]$ and $h :: t \in [\![s]\!]$, whereas the right-hand side only guarantees a witness for each conjunct separately, possibly two different elements of $[\![\nu]\!]$; the two are not obviously the same set.

$$
\begin{aligned}
[\![\partial_\nu !r]\!] &= \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!]\ h :: s \in [\![!r]\!]\} && \text{By Def 2.21} \\
&= \left\{s \in \Sigma^* \;\middle|\; \exists h \in [\![\nu]\!]\ h :: s \in \overline{[\![r]\!]}\right\} && \text{By Fig 2} \\
&= \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!]\ h :: s \notin [\![r]\!]\} && (2.14) \\
[\![!\partial_\nu\, r]\!] &= \overline{[\![\partial_\nu\, r]\!]} && \text{By Fig 2} \\
&= \overline{\{s \in \Sigma^* \mid \exists h \in [\![\nu]\!]\ h :: s \in [\![r]\!]\}} && \text{By Def 2.21} \\
&= \{s \in \Sigma^* \mid \forall h \in [\![\nu]\!]\ h :: s \notin [\![r]\!]\} && \text{De Morgan} \quad (2.15)
\end{aligned}
$$

In [New25b] we sketched a proof of Equation (2.8), which we complete here; it depends on the fact that any σFA can be determinized (Proposition 2.16).

Proposition 2.23 (derivative of complement)**.** *Let r be an RTE whose labels are among $\mu_1, \ldots, \mu_m$, and let $\nu_1, \ldots, \nu_n \in \Upsilon$ be designators whose types $[\![\nu_1]\!], \ldots, [\![\nu_n]\!]$ partition Σ and refine every μ_j; that is, for all i, j either $[\![\nu_i]\!] \subseteq [\![\mu_j]\!]$ or $[\![\nu_i]\!] \subseteq \overline{[\![\mu_j]\!]}$. Then for each $i \in \{1, \ldots, n\}$,*

$$[\![\partial_{\nu_i} !r]\!] = [\![!\partial_{\nu_i}\, r]\!]\,.$$

The proof uses a σCDFA A with $\mathcal{L}\{A\} = [\![r]\!]$; one exists by the construction of Section 2.9.

Proof. Fix i and write $\nu = \nu_i$. As illustrated in Figure 4, we build σCDFAs B and C from A with $B = \overrightarrow{A}_{q_1}$ and $C = \overline{A}$, then compute $\mathcal{L}\{D\}$ for $D = \overline{B} = \overrightarrow{C}_{q_1}$ in two ways.

(1) Since A is deterministic and q_0 has an outgoing transition $(q_0) \xrightarrow{\nu} (q_1)$, Equation (2.13) gives $\mathcal{L}\{\overrightarrow{A}_{q_1}\} = [\![\partial_\nu\, r]\!]$. We follow how this transition is transformed in constructing B, C, and D.

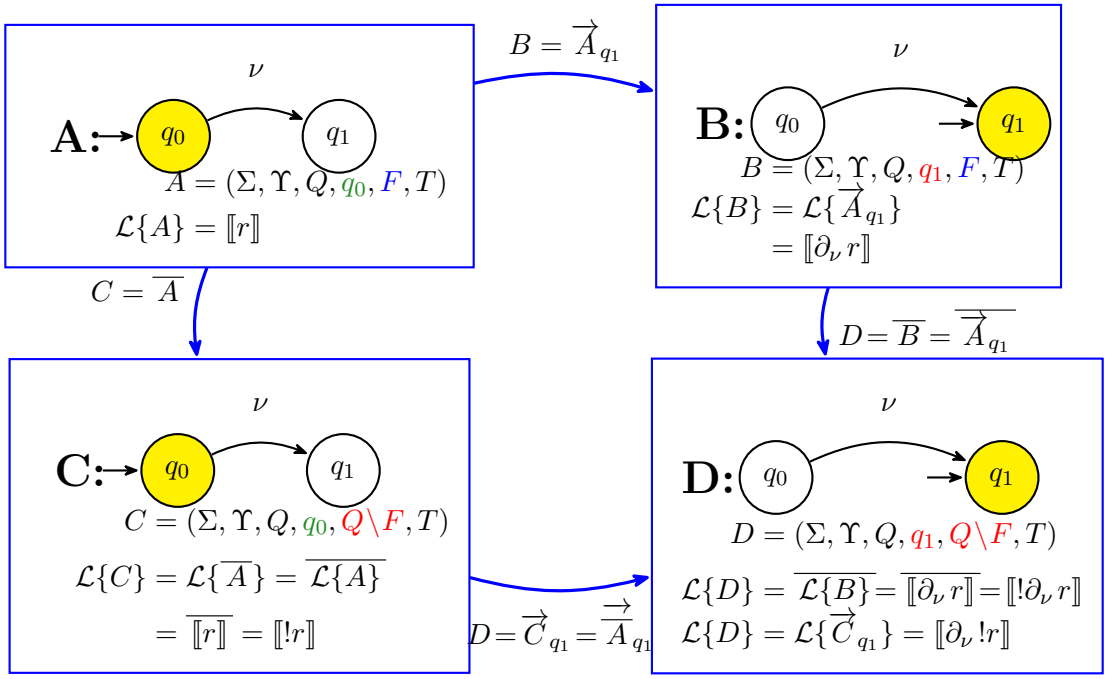


FIGURE 4. Transformations of σCDFA: A is transformed to produce B and C. D is constructed by transforming either B or C illustrating that $\overrightarrow{\overline{A}}_{q_1} = \overrightarrow{\overline{A}}_{q_1}$.

(2) **The right-automaton of A at q_1:** Let $B = \overrightarrow{A}_{q_1} = (\Sigma, \Upsilon, Q, q_1, F, T)$. B is complete (by Proposition 2.18). $\mathcal{L}\{B\} = \mathcal{L}\{\overrightarrow{A}_{q_1}\} = [\![\partial_\nu r]\!]$.

State q_0 may be non-accessible in B, but we don't care. State q_0 is preserved so that A and B have the same states and transitions, differing only by the initial state: q_1 vs q_0, thus assuring Equation (2.16). See Figure 4 (upper-right).

(3) **The complement automaton of A:** Let $C = \overline{A} = (\Sigma, \Upsilon, Q, q_0, Q \setminus F, T)$. Hence, $\mathcal{L}\{C\} = \mathcal{L}\{\overline{A}\} = \overline{\mathcal{L}\{A\}} = \overline{[\![r]\!]} = [\![!r]\!]$, (by Figure 2). A and C have the exact same states and transitions and differ only by the designation of final (accepting) states: $Q \setminus F$ vs F. See Figure 4 (lower-left).

(4) $\overrightarrow{\overline{A}}_{q_1}$ **vs** $\overrightarrow{\overline{A}}_{q_1}$**:** Finally, we remark that

$$\overline{B} = (\Sigma, \Upsilon, Q, q_1, Q \setminus F, T) = \overrightarrow{C}_{q_1}\,. \tag{2.16}$$

Thus $\overline{B} = \overrightarrow{C}_{q_1}$. So let $D = (\Sigma, \Upsilon, Q, q_1, Q \setminus F, T)$ and ask what $\mathcal{L}\{D\}$ is. See Figure 4 (lower-right).

Since D is the right-automaton of C at q_1, then $\mathcal{L}\{D\} = \mathcal{L}\{\overrightarrow{C}_{q_1}\} = [\![\partial_\nu !r]\!]$. However, D is simultaneously the complement of B, so $\mathcal{L}\{D\} = \overline{\mathcal{L}\{B\}} = \overline{[\![\partial_\nu r]\!]}$. But by Figure 2, $\overline{[\![\partial_\nu r]\!]} = [\![!\partial_\nu r]\!]$. Therefore, $[\![\partial_\nu !r]\!] = [\![!\partial_\nu r]\!]$. □

Equation (2.7), the intersection rule, follows from Proposition 2.23 at no extra cost: it is not itself primitive, since Figure 2 already gives $r \,\&\, s$ and $!(!r + !s)$ the same language by De Morgan.

Corollary 2.24 (derivative of intersection)**.** *Let r and s be RTEs whose labels are among $\mu_1, \ldots, \mu_m$, and let $\nu_1, \ldots, \nu_n \in \Upsilon$ be as in Proposition 2.23. Then for each $i \in \{1, \ldots, n\}$,*

$$[\![\partial_{\nu_i}(r \,\&\, s)]\!] = [\![\partial_{\nu_i} r \,\&\, \partial_{\nu_i} s]\!] .$$

Proof. Write $\nu = \nu_i$ and $q = !r + !s$. By Figure 2, $[\![r \,\&\, s]\!] = [\![!q]\!]$, so by Definition 2.21, $[\![\partial_\nu (r \,\&\, s)]\!] = [\![\partial_\nu\, !q]\!]$. Since ! introduces no labels, the labels of q are exactly those of r and s combined, so ν refines every label of q, and of r and s individually. Applying Proposition 2.23 to q,

$$[\![\partial_\nu\, !q]\!] = [\![!\partial_\nu\, q]\!] .$$

By Equation (2.6), which holds for any ν, $\partial_\nu\, q = \partial_\nu\, !r + \partial_\nu\, !s$; applying Proposition 2.23 once more, to r and to s separately,

$$[\![\partial_\nu\, q]\!] = [\![!\partial_\nu\, r + !\partial_\nu\, s]\!] .$$

Combining the last two displays and applying Figure 2's De Morgan identity once more,

$$[\![\partial_\nu\, !q]\!] = [\![!\big(!\partial_\nu\, r + !\partial_\nu\, s\big)]\!] = [\![\partial_\nu\, r \,\&\, \partial_\nu\, s]\!] .$$

Therefore $[\![\partial_\nu (r \,\&\, s)]\!] = [\![\partial_\nu\, r \,\&\, \partial_\nu\, s]\!]$. □

3. Deciding Vacuity by Shortest-Path Search

Section 2 developed two threads whose interaction drives the remainder of the paper. One thread concerns the type layer and the reasons subtype and inhabitation questions may yield the answer `dont-know` (Sections 2.3, 2.4, and 2.6). The other concerns symbolic automata whose transition labels are drawn from those same type designators. This section is where those threads meet. The central question is how indeterminacy at the type level propagates to the language of a symbolic automaton. In classical finite automata theory, vacuity and habitation are routine questions. For symbolic automata with semi-Boolean predicates they become three-valued questions, and the goal of this section is to show that they nevertheless reduce to a single shortest-path computation.

The technical core of this section is a single observation: for a σCDFA whose transitions carry semi-Boolean predicates, the three-valued vacuity question—*is the language inhabited, vacuous, or can we not tell?*—is a shortest-path problem. Once the problem is posed that way, the algorithm is textbook Dijkstra and the correctness argument is a one-line counting lemma. The work is in the posing: the distinction between a transition we can *prove* satisfiable and one whose satisfiability is *indeterminate* is not a qualitative obstacle requiring new machinery, but a *quantity*—an edge weight—and the boundary between the two verdicts is a *path-length threshold*. A human looking at a drawn σCDFA can often tell, just by understanding what an indeterminate transition's predicate actually does, that it is really satisfiable—even when the subtype procedure cannot prove it. A machine has no such insight: real satisfiability is exactly what the procedure could not decide, so it must treat every indeterminate transition the same way. Recognizing the three-valued vacuity question as weighted reachability is what lets an algorithm find a good witness anyway, without needing that insight, and what yields a minimal witness for free.

This is not the first time an equivalent three-way classification has come up unresolved. Years earlier, in the doctoral work behind the type system this paper builds on [New18, NVC17], habitation of a Common Lisp type represented as a binary decision diagram faced the same three verdicts over a different representation: a diagram collapsing to the

constant-false node is soundly determinate-uninhabited; a root-to-true path passing through no predicate looks determinate-inhabited, but only if every reduction that shaped that region of the diagram was itself determinate, since those reductions are backed by the same undecidable subtype procedure. Whether a clean predicate-free path was a genuine witness or a phantom left by an unprovable reduction was never settled, and the question was never written up. The shortest-path reduction below is, in that sense, a second attempt at this same problem—the first was for decision diagrams over a type lattice; this one is for σCDFA transitions.

3.1. **The vacuity problem, and why it is not routine.** Classical finite automata theory treats language emptiness as a two-valued question: a language is either empty or inhabited. Section 2.6, however, showed that the predicate reasoning underneath our transition labels may yield the answer `dont-know`. The first task, therefore, is to determine how that third outcome should be reflected at the automata level. Definition 3.1 introduces the distinction that drives the remainder of this section.

Recall (Definition 2.6) that a transition $\textcircled{q} \xrightarrow{\nu} \textcircled{r}$ is *satisfiable* exactly when $[\![\nu]\!] \neq \emptyset$. Under a semi-Boolean subtype procedure this property is not decidable, so we distinguish what the procedure *reports* from what is *true*.

Definition 3.1 (determined / indeterminate transition)**.** Fix a σCDFA A and its semi-Boolean satisfiability procedure. A transition $t = \textcircled{q} \xrightarrow{\nu} \textcircled{r}$ is

- *determined-satisfiable* if the procedure proves $[\![\nu]\!] \neq \emptyset$;
- *determined-unsatisfiable* if the procedure proves $[\![\nu]\!] = \emptyset$;
- *indeterminate* otherwise.

A computation (Definition 2.13) is *determined* if every one of its transitions is determined-satisfiable.

Determined-unsatisfiable transitions carry no words and are removed before any of what follows; "transition" below means determined-satisfiable or indeterminate. We seek a procedure that, given A, returns one of

- INHABITED: a proof that $\mathcal{L}\{A\} \neq \emptyset$, *together with a witness* $s \in \mathcal{L}\{A\}$;
- VACUOUS: a proof that $\mathcal{L}\{A\} = \emptyset$;
- INDETERMINATE: neither is provable.

and, because the procedure is meant to drive property-based testing (Section 4), we want the witness to be as *short* as possible: a minimal failing sequence is what a developer debugs.

Three routine attacks fall short.

Classical minimization. The textbook non-emptiness test for a finite automaton [HMU06] first prunes non-accessible and non-co-accessible states, after which the language is empty iff no accepting state remains. Pruning requires deciding which transitions are satisfiable; under semi-Boolean predicates we cannot, so we cannot reliably identify the states to prune. Our `minimize` function[4] (Section 4) still removes states it can *prove* dead and merges states whose outgoing labels simplify to identical designators—it shrinks the graph and speeds the search below without affecting any verdict—but it cannot reduce the question to "are there accepting states?". Internally, `minimize` is ordinary partition refinement: it refines the partition of states repeatedly until it is stable. That is Moore's algorithm [Moo56], not

[4]Named optimistically: under semi-Boolean predicates `minimize` cannot always reach, or certify that it has reached, the minimum, for the reasons given in this paragraph.

Hopcroft's asymptotically faster worklist algorithm [Hop71], although our implementations name the function after Hopcroft; both compute the same partition. The one symbolic adjustment is that, at each state, the labels of all transitions leading into the same class are first combined into a single designator, and two states stay in the same class only if their (designator, class) pairs coincide. The designators are compared by their simplified form, not by testing whether their difference is empty, so two states whose labels are equivalent, even provably so, but simplify differently are not merged: `minimize` is sound but not maximal. Newton and Verna [NV19] first noted this limitation for the original Common Lisp version of the algorithm; Section 5 proposes a stronger step. Concretely, two exiting transitions at the same state (Section 2.10) may both be provably disjoint yet both unprovably empty. Let `f` be `(fn [n] (and (> n 5) (< n 3)))`—never true, though the procedure does not inspect its arithmetic to see this—and consider `(and (satisfies f) (satisfies even?))` versus `(and (satisfies f) (not (satisfies even?)))`: disjoint by construction, and both genuinely empty since `f` alone already is, yet neither emptiness is provable from an opaque `satisfies`. `minimize` cannot tell they denote the same language, even though they trivially do.

Enumerating assumptions. One may branch on the true/false value of each indeterminate transition and test the 2^k resulting classical automata. This is exponential in the number of indeterminate transitions and produces no single witness.

Two independent searches. Search once using only determined-satisfiable transitions; if an accepting state is reached the language is INHABITED. Otherwise search again using all transitions; if an accepting state is now reached, every accepting computation must use an indeterminate transition, so the verdict is INDETERMINATE; if not, the language is VACUOUS. Read naively—taking whatever accepting state a traversal happens to reach first—this need not deliver a shortest witness; restricted to a breadth-first search of the determined subgraph specifically, it does, in $O(n+m)$ (Remark 3.6). What survives even that correction is that it is still two traversals, that it treats the inhabited/indeterminate boundary as a control-flow case split rather than as a property of a single quantity, and that it stops working once indeterminate transitions carry non-uniform cost (Section 3.3), where a genuine weighted search is needed and two passes no longer suffice. The reduction below collapses the two searches into one, trading the tighter two-pass bound below for that generality.

3.2. **A weighted-reachability reduction.** Let $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ be a σCDFA with $n = |Q|$ states and $m = |T|$ transitions. Assign to each transition the weight

$$\mathrm{w}(t) \;=\; \begin{cases} 1 & \text{if } t \text{ is determined-satisfiable,} \\ n & \text{if } t \text{ is indeterminate.} \end{cases}$$

Adjoin a fresh *goal* vertex $q_\top \notin Q$ and a weight-0 edge $\textcircled{f} \to \textcircled{q_\top}$ for every $f \in F$. The weight of a computation is the sum of its transition weights; let W^* be the least weight of any walk from q_0 to $q_\top$, with $W^* = \infty$ if no such walk exists.

Lemma 3.2 (length bound)**.** *If A has a determined accepting computation, then it has one with at most $n-1$ transitions, hence of weight at most $n-1$.*

Proof. Take a determined accepting computation. If some state occurs twice in it, excise the sub-computation strictly between the two occurrences; the result is again a computation (consecutive transitions still meet), still ends in the same accepting state, and uses a subset

of the original transitions, so is still determined. Repeating until no state repeats leaves a computation visiting at most n distinct states, hence using at most $n-1$ transitions, each of weight 1. This is the familiar pumping-lemma argument for DFAs [HMU06, Sak09], adapted to additionally track determinedness: pumping only ever removes transitions, never introduces an indeterminate one. □

The key idea is that indeterminacy can be represented by the weighting of transitions. The reduction assigns weight 1 to determined-satisfiable transitions and weight n to indeterminate ones. Lemma 3.2, above, establishes that every determined accepting computation lies below a natural threshold. Theorem 3.3, below, shows that comparing a shortest-path weight against that threshold is enough to distinguish inhabited, vacuous, and indeterminate languages.

Theorem 3.3 (vacuity by shortest path)**.** *With W^* as above:*

(1) $W^* = \infty$ *iff A has no accepting computation; then $\mathcal{L}\{A\} = \emptyset$ (*VACUOUS*).*
(2) *If $W^* < n$ then $\mathcal{L}\{A\} \neq \emptyset$ (*INHABITED*). A least-weight $q_0 \rightsquigarrow q_\top$ path is a determined accepting computation, and among all determined accepting computations it uses the fewest transitions; choosing one symbol per label yields a shortest witness.*
(3) *If $n \leq W^* < \infty$ then A has an accepting computation but no determined one: every accepting computation traverses at least one indeterminate transition. The verdict is* INDETERMINATE*.*

The three cases are exhaustive and mutually exclusive, and W^* is computable by one shortest-path search in $O(m + n \log n)$ time.

Proof. A $q_0 \rightsquigarrow q_\top$ walk is a computation from q_0 to some $f \in F$ followed by the 0-weight goal edge; such walks correspond exactly to accepting computations of A. This gives (1): $W^* = \infty$ iff there is no accepting computation, and with no accepting computation no word is accepted, so $\mathcal{L}\{A\} = \emptyset$.

For (2) and (3), note every transition weight is 1 or n, so a walk of weight $\leq n-1$ contains no indeterminate transition and is therefore determined. If $W^* \leq n-1$, a least-weight walk is a determined accepting computation $(\nu_1, \ldots, \nu_k)$; each $[\![\nu_i]\!] \neq \emptyset$, so picking $s_i \in [\![\nu_i]\!]$ gives $(s_1, \ldots, s_k) \in \mathcal{L}\{A\}$. By Lemma 3.2, if any determined accepting computation exists then $W^* \leq n-1$; conversely we just saw $W^* \leq n-1$ forces one to exist. Hence case (2) holds *exactly* when a determined accepting computation exists, and then W^* equals the minimum number of transitions over all such computations, so the recovered witness is shortest.

Case (3) is the complement within $W^* < \infty$: an accepting computation exists, but—by the equivalence just shown—no determined one, so every accepting computation uses an indeterminate transition. We cannot certify inhabitation (no determined accepting computation, and an indeterminate transition may be unsatisfiable) and cannot certify vacuity (an accepting computation exists, and its indeterminate transitions may all be satisfiable); the verdict is INDETERMINATE.

Exhaustiveness and exclusivity follow because the three predicates $W^* = \infty$, $W^* \leq n-1$, $n \leq W^* < \infty$ partition the possible values of W^*: no walk weight lies in $[\,n-1,\ n\,)$ other than $n-1$ itself, and a weight of exactly $n-1$ is realisable only by an all-determined walk, already covered by (2). Dijkstra's algorithm [Dij59] computes W^* (and the path) with non-negative weights in $O(m + n \log n)$. □

The significance of Theorem 3.3 is that it reduces the entire three-valued vacuity question to the computation of a single quantity, the shortest-path weight W^*. The three possible

verdicts—inhabited, vacuous, and indeterminate—arise from comparing that quantity against a threshold determined by the size of the automaton. The theorem also yields a shortest witness whenever inhabitation can be certified, making the result useful not only for decision procedures but also for the testing applications developed in Section 4.

Algorithm 3.1 states the procedure of the proof step by step; it calls the semi-Boolean satisfiability procedure once per transition. Our `dfa-inhabited?` is built on the same shortest-path search and returns the three-way verdict; the witness is read from the path that search finds.

Input: a σCDFA $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ with $n = |Q|$
Input: *sat*: semi-Boolean satisfiability procedure, returning *true*, *false*, or `dont-know`
Output: VACUOUS; INHABITED with a shortest witness; or INDETERMINATE

3.1.1 **begin**
3.1.2 $E \leftarrow \emptyset$ `// weighted edges of the search graph`
3.1.3 **foreach** $(p, \nu, q) \in T$ **do**
3.1.4 $v \leftarrow sat(\nu)$;
3.1.5 **if** $v = true$ **then**
3.1.6 $E \leftarrow E \cup \{(p, q, 1)\}$ `// determined-satisfiable`
3.1.7 **else if** $v =$ *`dont-know`* **then**
3.1.8 $E \leftarrow E \cup \{(p, q, n)\}$ `// indeterminate`
`//` $v =$ *`false`*`: determined-unsatisfiable, dropped`
3.1.9 **foreach** $f \in F$ **do**
3.1.10 $E \leftarrow E \cup \{(f, q_\top, 0)\}$ `// fresh goal vertex` $q_\top$
3.1.11 $(W^*, \pi) \leftarrow Dijkstra(E, q_0, q_\top)$ `//` $W^* = \infty$ `if` $q_\top$ `is unreachable`
3.1.12 **if** $W^* = \infty$ **then**
3.1.13 **return** VACUOUS
3.1.14 **else if** $W^* < n$ **then**
`//` π `uses only weight-1 transitions, labeled` $\nu_1, \ldots, \nu_k$
3.1.15 **return** INHABITED, $(s_1, \ldots, s_k)$ with each $s_i \in [\![\nu_i]\!]$
3.1.16 **else**
3.1.17 **return** INDETERMINATE

Algorithm 3.1: Three-valued vacuity check by shortest path (Theorem 3.3)

Remark 3.4 (a strong-Kleene reading of the three cases)**.** Theorem 3.3's case split is not merely analogous to strong-Kleene three-valued logic; it is that logic, restated. Assign each transition t that actually enters the weighted graph above a value $\mathrm{sat}(t) = true$ if $\mathrm{w}(t) = 1$ and $\mathrm{sat}(t) =$ `dont-know` if $\mathrm{w}(t) = n$; unsatisfiable transitions take no part in this graph at all (Remark 2.22), consistent with assigning them *false* and letting *false* mean "not an edge here." Under strong-Kleene semantics—conjunction is the minimum and disjunction the maximum, under the order *false* < `dont-know` < *true*—the three cases of the theorem are exactly the three possible values of

$$\bigvee_{c \text{ accepting}} \bigwedge_{t \in c} \mathrm{sat}(t),$$

disjunction over accepting $q_0 \rightsquigarrow q_\top$ walks c, conjunction over c's own transitions. A walk's conjunction is *true* exactly when every transition is determined-satisfiable—a determined accepting computation—and `dont-know` as soon as one transition is, since no transition in this graph is ever *false*. The outer disjunction is then *true* if some determined accepting computation exists (case 2), `dont-know` if an accepting computation exists but none is determined (case 3), and *false*—the empty disjunction—if no accepting computation exists at all (case 1).

The weighting has a reading in the same terms. Weight 1 for a *true* transition and n for a `dont-know` one makes every all-*true* accepting computation strictly cheaper, at most $n-1$ by Lemma 3.2, than any computation using even one `dont-know` transition. A shortest-path search therefore always returns a *true*-valued computation when one exists, and only reports weight $\geq n$ once it must fall back to a `dont-know`-valued one—exactly Kleene disjunction's own priority, *true* before `dont-know` before absence, encoded as a minimization objective. This is why the reduction works, not merely that it does.

Remark 3.5 (the oracle, not the reduction, is the bottleneck)**.** Theorem 3.3's $O(m+n\log n)$ is the cost of the graph search alone. Classifying each of the m transitions as determined-satisfiable or indeterminate costs one call to the host's satisfiability procedure, at some cost C per call that this paper leaves unspecified—cheap for a hosted-type test, potentially expensive for an opaque predicate (Section 2.3). The reduction adds only $O(m+n\log n)$ on top of the m oracle calls any approach, ours included, must pay regardless; it is not itself the bottleneck.

Remark 3.6 (a linear two-pass alternative for the binary case)**.** The two-pass search of Section 3.1 *can* be made to return a shortest witness, in $O(n+m)$—strictly better than Theorem 3.3's $O(m+n\log n)$—if its first pass is a plain, unweighted breadth-first search restricted to the determined-satisfiable subgraph. By Lemma 3.2, every all-determined accepting computation lies entirely inside that subgraph, so the shortest one this restricted search finds to any accepting state is the global minimum over all determined accepting computations—exactly what Theorem 3.3 case (2) returns, with no weights needed. A second, plain reachability pass over the full graph, run only if the first finds nothing, then distinguishes INDETERMINATE from VACUOUS, also in $O(n+m)$. We nonetheless prefer the single weighted search: the paragraph below shows the same threshold argument survives verbatim once indeterminate transitions carry different costs, where this two-pass construction no longer applies.

3.3. **What the reduction buys.** Beyond its correctness proof, the reduction of Theorem 3.3 yields structural advantages that matter in practice, for witness generation, testing, and extensions of the basic model.

One quantity, one search. The entire three-valued decision is the comparison of a single number W^* against the single threshold n. There is no case analysis over which transitions "really" exist and no second traversal.

A minimal witness, for free. Because determined transitions have unit weight, a least-weight accepting path is a fewest-transition accepting path: the witness Dijkstra returns is already the shortest failing sequence, which is exactly what property-based testing wants to hand to a developer (Section 4). The two-pass search does not give this without extra work.

Soundness is robust to a weak oracle. Reclassifying a transition from determined-satisfiable to indeterminate only raises weights, so it can only move a verdict in the direction INHABITED $\to$ INDETERMINATE $\to$ VACUOUS; it can never manufacture a false INHABITED. A more conservative satisfiability procedure therefore never makes the habitation check unsound, only less informative.

The binary case is cheap; the graded case is the same algorithm. With only two weight classes, $W^* < n$ versus $W^* \geq n$ can be decided by one breadth-first search of the determined subgraph (returning the shortest witness) and, on failure, one of the full graph—$O(m+n)$, already stated precisely as Remark 3.6. We nonetheless state the reduction as weighted reachability because the threshold argument survives verbatim when indeterminate transitions carry *different* costs—for instance the number of indeterminate predicates a label depends on, or an estimated cost to resolve it—where one Dijkstra search then returns the accepting computation of least total indeterminacy, still with a witness.

An under-/over-approximation reading. Let A^- keep only the determined-satisfiable transitions of A and A^+ keep the determined-satisfiable and indeterminate ones. Then $\mathcal{L}\{A^-\} \subseteq \mathcal{L}\{A\} \subseteq \mathcal{L}\{A^+\}$: A^- under-approximates and A^+ over-approximates the language. Theorem 3.3 is exactly the simultaneous evaluation of both bounds by one search—$W^* \leq n-1$ witnesses $\mathcal{L}\{A^-\} \neq \emptyset$, $W^* = \infty$ witnesses $\mathcal{L}\{A^+\} = \emptyset$, and the gap between the two, $n \leq W^* < \infty$, is the region where the semi-Boolean predicate's third truth value propagates to the language.

—FOR AUTHOR'S APPROVAL—

The bracketing is a Galois connection. Read each transition's epistemic state as a nonempty set of possible truth values: $\{\mathit{true}\}$ when determined-satisfiable, and $\{\mathit{true}, \mathit{false}\}$ when `dont-know`—determined-unsatisfiable transitions are already pruned before any of this (Remark 2.22), so $\{\mathit{false}\}$ alone never arises here. Concretization γ sends $\mathit{true} \mapsto \{\mathit{true}\}$, `dont-know` $\mapsto \{\mathit{true}, \mathit{false}\}$, and, for completeness, $\mathit{false} \mapsto \{\mathit{false}\}$; abstraction α returns the sharpest label consistent with a given set of possible truth values. This pair is a Galois connection between the powerset of $\{\mathit{true}, \mathit{false}\}$, ordered by $\subseteq$, and the three labels, ordered $\mathit{false} \sqsubset$ `dont-know` and $\mathit{true} \sqsubset$ `dont-know`: $\alpha(S) \sqsubseteq a \iff S \subseteq \gamma(a)$ for every concrete S and label a. A^- and A^+ are exactly γ applied pointwise to each transition's label, split by "concretization is exactly $\{\mathit{true}\}$"—the *must* reading—versus "*true* is a member of the concretization"—the *may* reading. Since *false*-labeled transitions are already pruned, the *may* reading keeps every transition that remains, so $A^+ = A$: no separate construction is needed, only this observation. This is the standard must/may soundness pattern from abstract interpretation, and specifically the instance of it, K_3 as a Galois-connection abstraction of the Booleans, that underlies three-valued abstract interpretation generally—most prominently TVLA, parametric shape analysis via 3-valued logic [SRW02]. We are not claiming our automata are instances of TVLA's shape-analysis structures, only that the same well-known soundness pattern is at work here.

—FOR AUTHOR'S APPROVAL—

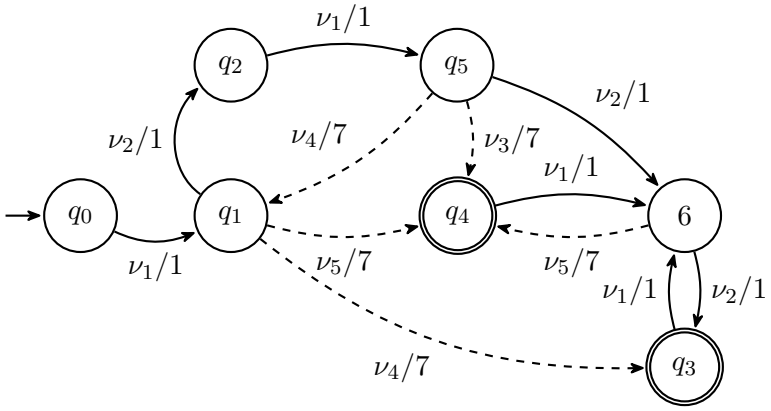


FIGURE 5. A σCDFA as a Dijkstra-compatible weighted graph. Solid edges (weight 1) are determined-satisfiable; dashed edges (weight $n = 7$) are indeterminate. The sink state and its incoming edges are omitted.

A well-known algorithm is easier to trust than a bespoke one. The weighted-reachability idea itself predates this theorem. Our Scala implementation first cast habitation this way in 2022, weighting determined-satisfiable transitions 1 and indeterminate ones large enough that no all-determined path could lose to one using them—using Bellman–Ford, which assumes nothing about the sign of a weight. Nearly two years later a genuine bug surfaced in its hand-written path reconstruction, a broken recursive step that silently returned a one-hop prefix instead of the full path; fixing it and noting that every weight here is already non-negative motivated switching the default search to Dijkstra—provably correct and asymptotically better. Our Clojure implementation took a different route for longer: its habitation check called a hand-rolled, mutually-recursive path enumerator twice—once disallowing and once allowing indeterminate labels, exactly the two-pass attack above—and carried its own unresolved `TODO` admitting it did not yet handle indeterminate labels correctly, until it too converged on a single Dijkstra search in 2025, independently unit-tested and decoupled from any automaton-specific code, matching the design Scala had already settled on. Reducing the problem to a well-known, independently specifiable algorithm did not just simplify either implementation; it made the search itself verifiable in a way neither bespoke version had been.

3.4. **Example.** Consider the σCDFA of Figure 5 ($n = 7$; the sink state and its incoming transitions are omitted). Solid arrows are determined-satisfiable (weight 1), dashed arrows are indeterminate (weight 7); ν_3, ν_4, ν_5 are indeterminate.

The accepting state q_3 is reached by the determined computation $(\nu_1, \nu_2, \nu_1, \nu_2, \nu_2)$ of weight $5 \leq n - 1$, so $W^* = 5$ and the language is INHABITED with a witness of length 5. Had we instead removed the two known-satisfiable loops, the only route to the other accepting state q_4 would be (ν_1, ν_5) of weight $8 \geq n$, or $(\nu_1, \nu_2, \nu_1, \nu_3)$ of weight 10; a graph in which q_4 were the sole accepting state would return $W^* = 8$ and the verdict INDETERMINATE. The weight-8 computation may in fact be unsatisfiable—by Definition 3.1 we cannot tell—which is precisely why the verdict is INDETERMINATE rather than INHABITED.

3.5. **Naming.** At the risk of redundancy for some readers: a *habitation check* attempts to prove a set is inhabited (non-empty); a *constructive* habitation check exhibits a member; a *vacuity check* proves a set empty, and can often be run as a habitation check with reversed parity. Theorem 3.3 does all three at once.

4. Testing Symbolic Automata Libraries by Differential Oracles

Sections 2 and 3 developed the symbolic-automata theory that underlies this paper's three-valued vacuity check. We now return to the practical setting that motivated much of that development. The connection is direct: each testing oracle described in this section ultimately reduces a language-equivalence question to a vacuity question, and therefore depends on Theorem 3.3. This section explains how that theorem is used in practice and what kinds of defects the resulting oracles can detect and fail to detect.

We maintain three independent implementations of the same theory—σCDFA libraries in Scala, Python, and Clojure [New24d, New24c, New24a, New25b] (Section 3 discusses a fourth, earlier Common Lisp implementation [New24b] that predates this theory's semi-Boolean extension and is not part of this comparison). The testing question is how to turn that redundancy into evidence of correctness.

A test *oracle* is the part of a test that decides whether an observed output is correct—the expected value in a hand-written assertion, a trusted reference implementation, or an invariant the output must satisfy [BHM+15]; for randomly generated input, where no expected value can be written down in advance, the oracle is necessarily such an invariant. This section describes three oracles of increasing power. The weakest, a round-trip equivalence check on a single implementation, is the one the earlier version of this paper reported; we now make its blind spots precise and show that the bugs it misses are found either by pinning each operation to its specification, or by comparing the three implementations against one another.

4.1. **Equivalence through the difference automaton.** The testing methodology developed in this section is concerned with language equivalence: whether two independently constructed symbolic automata recognize the same language. Theorem 3.3, however, answers a vacuity question rather than an equivalence question. Proposition 4.1 supplies the connection between those two viewpoints. It reduces equivalence to vacuity and therefore allows every oracle in this section to reuse the machinery developed in Section 3.

Every oracle below rests on the same identity.

Proposition 4.1. *Two σCDFAs, A and B, express the same language if and only if $\mathcal{L}\{A\} \oplus \mathcal{L}\{B\}$ is empty.*

Proof.

$$\begin{aligned}
\mathcal{L}\{A\} = \mathcal{L}\{B\} &\iff (\mathcal{L}\{A\} \subseteq \mathcal{L}\{B\}) \wedge (\mathcal{L}\{B\} \subseteq \mathcal{L}\{A\}) \\
&\iff \left(\mathcal{L}\{A\} \cap \overline{\mathcal{L}\{B\}} = \emptyset\right) \wedge \left(\mathcal{L}\{B\} \cap \overline{\mathcal{L}\{A\}} = \emptyset\right) \\
&\iff \left(\mathcal{L}\{A\} \cap \overline{\mathcal{L}\{B\}}\right) \cup \left(\mathcal{L}\{B\} \cap \overline{\mathcal{L}\{A\}}\right) = \emptyset \\
&\iff \mathcal{L}\{A\} \oplus \mathcal{L}\{B\} = \emptyset
\end{aligned}$$

□

Given a σCDFA for $\mathcal{L}\{A\} \oplus \mathcal{L}\{B\}$, its vacuity is decided (or found indeterminate) by Theorem 3.3. Under semi-Boolean predicates "$\mathcal{L}\{A\} \oplus \mathcal{L}\{B\}$ is empty" is itself sometimes only INDETERMINATE; an oracle may then conclude nothing, but it must never report a spurious disagreement.

4.2. **Oracle A: round-trip equivalence.** Oracle A exercises one implementation end to end, using property-based testing [FB97, CH00]: rather than hard-coded cases it asserts an invariant on randomly generated input. The invariant is that the following round trip preserves the language.

(1) Generate a σCDFA A with `path-seeded` and `minimize` (Section 4.8).
(2) Extract an RTE r from A with `dfa-to-rte` [Sak09, Sec 2.4.2].
(3) Rebuild a σCDFA B from r with `rte-to-dfa` (Section 2.9).
(4) Form the difference σCDFA $A \oplus B$ with `synchronized-xor` [NV19].
(5) Check the vacuity of $\mathcal{L}\{A \oplus B\}$ with `dfa-inhabited?` (Section 3).

If step 5 reports $\mathcal{L}\{A \oplus B\}$ *inhabited*, some function in the pipeline is wrong, and the habitation witness (Theorem 3.3) is a failing input sequence. The functions under test are `path-seeded`, `minimize`, `dfa-to-rte`, `rte-to-dfa`, `synchronized-xor`, and `dfa-inhabited?`; step 2 follows the classical state-elimination method and is not itself a contribution.

Oracle A is cheap and fully automatic, but it is one-sided in three ways that matter.

Blind spot 1: generator coverage. `path-seeded` (Algorithm 4.1) first lays a simple path through all n states and only then adds further transitions, so every state it produces has out-degree at least one. Automata containing a state with *no* outgoing transition are never generated, and neither are several other structural families. A defect that only manifests on an excluded shape is invisible to Oracle A no matter how many samples are drawn.

Blind spot 2: monotone masking.

Proposition 4.2. *Let $\widehat{\oplus}$ and $\widehat{inh}$ be the implementations of the difference and the vacuity check. If, on a given run with $\mathcal{L}\{A\} = \mathcal{L}\{B\}$,*

$$\mathcal{L}\{\, A \,\widehat{\oplus}\, B \,\} \;\subseteq\; \mathcal{L}\{A\} \oplus \mathcal{L}\{B\} \;=\; \emptyset \quad \textit{or} \quad \widehat{inh} \textit{ under-reports habitation,}$$

then Oracle A reports PASS*, whatever errors $\widehat{\oplus}$ or $\widehat{inh}$ contain.*

Proof. Immediate: Oracle A fails only when step 5 certifies an accepting, satisfiable computation of $A \,\widehat{\oplus}\, B$. If $\mathcal{L}\{A \,\widehat{\oplus}\, B\,\}$ is empty no such computation exists; if $\widehat{inh}$ under-reports it is not found. □

Any bug whose net effect on the pipeline is to *remove* behaviour from the difference automaton therefore raises Oracle A's confidence rather than lowering it.

Blind spot 3: symmetric masking. `dfa-to-rte` and `rte-to-dfa` share primitives—subtype and disjointness tests, label canonicalization, minterm construction. An error in a shared primitive that the round trip inverts (extract then rebuild) leaves $\mathcal{L}\{B\} = \mathcal{L}\{A\}$ and Oracle A silent, even though a direct client of that primitive would see the wrong answer.

TABLE 1. The `complete` defect across implementations.

implementation	states to repair enumerated by	out-degree-0 state
Clojure	state set	repaired
Python	state set	repaired
Scala	transitions grouped by source	**missed**

4.3. **Oracle B: operations against their specification.** Oracle B drops the round trip. For a fixed, deliberately adversarial corpus of RTEs $\{r_i\}$ and sequences $\{s_j\}$, and for each Boolean operation $op \in \{\cup, \cap, \oplus, \setminus, \ldots\}$, it checks the pointwise identity

$$simulate(op(A_i, A_j),\ s) \ =\ op_{\mathbb{B}}(simulate(A_i, s),\ simulate(A_j, s)) \tag{4.1}$$

where $A_i = \texttt{rte-to-dfa}(r_i)$ and $op_{\mathbb{B}}$ is the corresponding Boolean connective on the two membership verdicts. Equation (4.1) pins each product construction directly to its definition; it does not depend on `dfa-to-rte`, and it is *not* monotone-masked, because a product automaton that has lost paths will disagree with $op_{\mathbb{B}}$ on some s.

4.4. **Oracle C: cross-implementation agreement.** Oracle C parses one surface RTE in each host language and compares the implementations: membership verdicts on a shared sequence corpus, minimized state counts, and the equivalence (Proposition 4.1) of the extracted RTEs. A divergence localises a defect to at most one implementation—the strongest signal of the three, and the one the follow-on work develops by having a language model read the corresponding routines of all four implementations side by side.

4.5. **Findings.** The three oracles differ not only in how they test the implementations but also in the kinds of defects they are capable of exposing. Oracle A primarily tests the internal consistency of a single implementation, whereas Oracles B and C tie operations more directly to their specifications or to independently developed implementations. The findings below illustrate the practical consequences of that difference: some defects are visible to all three approaches, while others remain invisible until a stronger oracle is brought to bear.

`path-seeded` non-determinism (Oracle A).. Oracle A did find one genuine defect autonomously: `path-seeded` occasionally emitted two transitions from one state with overlapping labels, violating determinism. The bug is in the generator, not the library, but it is a real property-based-testing catch and it is why Algorithm 4.1 now subtracts the already-chosen labels of a state before adding a transition.

`complete` in one implementation of three (Oracle B; Oracle C suffices). `complete` makes a σCDFA complete by adding to each state a transition to a sink state, labelled by the complement of the union of that state's existing outgoing labels (Proposition 2.10). The Scala implementation enumerated the states to repair by grouping the *transition* relation by source; a state with no outgoing transition is absent from that grouping, is never repaired, and remains incomplete. The Clojure and Python implementations enumerate the *state set* and are unaffected (Table 1).

On a σCDFA considered alone the omission is harmless: an out-degree-0 non-accepting state behaves as a non-accepting sink. But `synchronized-xor` assumes both operands are complete; for a product state (q_1, q_2) it intersects the outgoing labels of q_1 and q_2, so if q_1

has none, every product state $(q_1, \cdot)$ is a dead end and whole families of computations vanish from the difference automaton. By Proposition 4.2 this is exactly the kind of bug Oracle A cannot see: the difference automaton only *shrinks*. Oracle B caught it, with the minimal counterexample $r_1 = \Sigma^* \cap \texttt{Int}^*$, $r_2 = \overline{\texttt{(= 1)}} \cap \texttt{Int}$, $s = (2, 3, 4)$: here $s \in \mathcal{L}\{r_1\}$, $s \notin \mathcal{L}\{r_2\}$, so $\mathit{simulate}(A_1 \cup A_2, s)$ must be true, but the incompletely-completed union returned false. Oracle C would have flagged the same run as a Scala-only disagreement. A companion defect in the same construction—exit-value arbitration reading the state identifiers of the *pre*-completion operands—was found and fixed at the same time.

A confidence result, not a bug count. Run on the mature libraries, Oracle A discovered no further discrepancy over 5040 generated σCDFAs. We read this as what it is: evidence that the round trip is self-consistent on the inputs the generator can reach, bounded by the three blind spots above—not as evidence that the libraries are correct. The defects that did surface required an oracle that either bypasses the round trip (B) or steps outside a single implementation (C).

4.6. **A second application: dead-code detection.** Up to this point, Theorem 3.3 has appeared primarily as the foundation of the testing methodology developed in this section. A reader could reasonably conclude that the theorem is valuable mainly because of that application. The purpose of this subsection is to show that the same reduction arises independently in a different setting. Dead-code detection in pattern-dispatch macros reduces to the same three-valued vacuity question, and therefore relies on the same shortest-path construction for reasons unrelated to testing. The example below is intentionally structural: its purpose is to illustrate the reduction itself. Section 4.7 then reintroduces types and shows how the theorem's three-way verdict becomes essential once reachability depends on subtype and inhabitation questions that may yield `dont-know`.

The testing methodology above is not Theorem 3.3's only consumer. The dispatch-macro scenario that opened this paper (Section 1.1) is real, working code, not a hypothetical: our Clojure implementation's `dsdefn` macro [New25a]—one of several variants in the same family, all sharing this mechanism—takes several `(pattern, body)` clauses and warns, at macro-expansion time, about any clause no call can ever reach. The mechanism *is* the theorem, not an independent reimplementation of the idea: the clauses' patterns are unioned into one σCDFA with each accepting state tagged by the clause it belongs to (`clauses-to-dfa`); one Dijkstra search is run from the initial state, using exactly the paper's own weighting—1 for determined-satisfiable transitions, n for indeterminate ones—to find, for every tag, the shortest accepting computation reaching it (`find-spanning-map`); a clause with no entry in the resulting map has no accepting computation at all, i.e. it is case (1) of Theorem 3.3, and only these clauses are flagged. Clauses reachable only via an indeterminate computation are left silent, a deliberately sound-for-warnings policy that the theorem's three-way verdict makes possible: it would not be expressible against a classical, two-way reachable/unreachable test. For example,

```
(dsdefn f
  ([[a b] c d] 12)
  ([a [b c] d] 13)
  ([a b [c d]] 14)
  ([[a b] c [d]] (println "fourth clause")
                 15))
;; prints to *err*, but throws no Exception
;;    Unreachable code: ((println "fourth clause") 15)
```

The fourth clause's pattern is syntactically distinct from all three before it—unlike the other three, it is not a duplicate—yet it can still never fire: its third position, `[d]`, matches only a one-element sequence, while the first clause leaves its own third position, `d`, unconstrained, so every call the fourth clause could match, the first clause already matches too. The fourth clause is subsumed by the first, not merely repeated by it, which is a genuinely different question than syntactic equality and one only the automaton answers correctly. We confirmed this directly against `find-spanning-map` rather than by inspection: of the four clauses, exactly the fourth is unreachable.

4.7. **Dead-code detection with types.** Section 4.6 illustrated dead-code detection using only the structure of patterns. That example established a second application of Theorem 3.3, but it did not yet exercise the semi-Boolean reasoning that motivates this paper. Introducing types changes the situation. Reachability may now depend not only on pattern structure but also on subtype and inhabitation questions that may yield the answer `dont-know`. The examples in this subsection show how the theorem's three-way verdict distinguishes provably unreachable code from code whose reachability remains indeterminate.

The `dsdefn` example above is purely structural: its patterns declare no types, and its dead clause is dead because of pattern shape alone. Types bring in the semi-Boolean procedure, and with it the full three-way verdict of Theorem 3.3. Extend the `classify` function of Section 1.1 with two more clauses:

```
(defn classify [& args]
  (destructuring-case args
    [[a b]      {a Long, b Long}]              :two-integers
    [[a [b]]    {a Long, b String}]            :integer-then-string
    [[a b]      {a (satisfies even?), b Long}] :even-then-integer
    [[a b]      {a (and Long String)}]         :impossible
    [[a & more] {a String}]                    :string-first))
```

At macro-expansion time this prints a single warning, `Unreachable code:  :impossible`, and nothing else.

A clause dead by type reasoning. The fourth clause requires `a` to be both a `Long` and a `String`. These classes are disjoint, so the procedure proves the label empty, the clause has no accepting computation (case 1 of Theorem 3.3), and the macro warns.

A clause that looks dead but is not. The third clause appears to be shadowed by the first: `(classify 4 5)` returns `:two-integers`, since every `Long` is caught by the first clause. But `even?` also accepts Clojure's arbitrary-precision integers, which are not `Long`, and `(classify 4N 5)` returns `:even-then-integer`. The procedure cannot see into `even?`, so every computation reaching the third clause passes through an indeterminate transition (case 3), and the macro says nothing. This is the sound-for-warnings policy of Section 4.6

at work: a two-valued checker that equated "no witness found" with "dead" would have warned about live code.

Our Scala implementation offers the identical check, computed by the same underlying machinery, but delivered differently. Here is `classify` again, with `rteIfThenElse`:

```
val int   = SAtomic(classOf[Int])
val str   = SAtomic(classOf[String])
val even  = SSatisfies(isEven, "even")
val oneStr = SSatisfies(isOneString, "one-string-seq")
val classify = rteIfThenElse(Seq(
    Cat(int, int)           -> (() => "two-integers"),
    Cat(int, oneStr)        -> (() => "integer-then-string"),
    Cat(even, int)          -> (() => "even-then-integer"),
    Cat(SAnd(int, str), Sigma) -> (() => "impossible"),
    Cat(str, Star(Sigma))   -> (() => "string-first")),
  () => "no match",
  handleUnreachable = rte => println(s"Unreachable: $rte"))
```

Patterns are built from constructors rather than written as lambda lists; no variables are bound, so a clause body cannot refer to `a` or `b`; the nested `[b]` of the second clause has no pattern syntax and must be written as a hand-coded predicate, `isOneString`; and the diagnostic, which fires when `rteIfThenElse` builds the dispatcher, at ordinary run time and only if the caller supplies a callback, reports an internal RTE rather than a source position. The verdicts are the same: only the fourth clause is reported, and `List(BigInt(4), 5)` reaches the third.

There is no automatic compile-time diagnostic, because `rteIfThenElse` simply is not implemented as a Scala macro at all—not because Scala lacks the means. Scala's own macro system could attach the same warning to a source position that a Clojure macro does, though a Scala macro must operate on a typed intermediate representation of the program rather than plain syntax, which is a substantially larger undertaking to author than a Lisp macro's syntax-in, syntax-out style. The concrete reason this project did not build one is more specific than that general difficulty: when `scala-rte` was first conceived, Scala macros were a Scala 2 feature, highly experimental and changing from release to release—not a stable foundation for a user-facing diagnostic. The codebase has since been migrated from Scala 2 to Scala 3, but its authors still have no expertise in Scala macros; informal conversations with Scala experts suggest that even with that expertise, a Lisp macro's flexibility may not be fully reachable in Scala. Closing that gap experimentally would be interesting, but it is future work, not currently planned.

Whichever host delivers it, the diagnostic is a warning, never an error: raising an exception would abort compilation of the whole file over what is meant to be advisory, and since the check is already sound—it never accuses reachable code of being dead—a false positive costs a nuisance warning, not a broken build. Finding a second, independent, already-implemented use for the same three-way reduction is stronger evidence that it matters than either application on its own.

4.8. **Generating non-trivial random σCDFAs.** The value of Oracles A and B depends entirely on the corpus. Our first generator built a random RTE abstract syntax tree—each internal node a uniformly chosen operator among concatenation, union, intersection, and Kleene star—and constructed the σCDFA from it. The syntax was uniform but the

languages were not: after minimization most σCDFAs had one or two states. Koechlin and Rotondo [KR21] explain why: with absorbing operators ($r \cap \emptyset = \emptyset$, $r \cup \Sigma = \Sigma$), a uniformly random tree collapses to a trivial language with high probability, so uniformity of syntax is not uniformity of semantics.

We reported an engineering workaround before; we have since replaced it with two structure-first generators, described and evaluated in companion talks [NZ25a, NZ25b]: a *tree-split* generator that fixes a leaf count and recursively partitions it by a biased pivot (Gaussian, inverse-Gaussian, linear, or lopsided), and a *Flajolet skeleton* generator that grows a random binary-search-tree shape and fits an RTE to it. Both target *portrait* rather than *landscape* aspect ratios—deep enough that the induced σCDFA is large after minimization—and both markedly reduce the fraction of trivial samples. Algorithm 4.1 gives the state-and-transition layer that sits on top of a chosen RTE generator.

Input: n: number of states; m: number of transitions
Input: *gentype*: type-designator generator honouring `type-size` and `probability-indeterminate`
Output: a σCDFA with initial state q_0 and single accepting state q_{n-1}

4.1.1 **begin**
4.1.2 $T \leftarrow []$;
`// states, joined by a simple path` $q_0 \to q_1 \to \cdots \to q_{n-1}$
4.1.3 **for** $i \leftarrow 1 \ldots n-1$ **do**
4.1.4 $\nu \leftarrow \mathit{gentype}()$;
4.1.5 $T \leftarrow (q_{i-1}, \nu, q_i) :: T$
4.1.6 **for** $i \leftarrow 1 \ldots m-n$ **do**
`// an extra forward, backward, or self-loop edge`
4.1.7 $(p, q) \leftarrow (\mathit{rand}(n-1), \mathit{rand}(n-1))$;
`// keep the new label disjoint from` q`'s existing outgoing labels`
4.1.8 $\nu \leftarrow \mathit{gentype}() \setminus \bigcup_{\mu \in \xi_A(q)} \mu$;
4.1.9 $T \leftarrow (q, \nu, p) :: T$

Algorithm 4.1: Compute states and transitions of a random σCDFA

The parameters are **num-states** (n), **num-transitions** (m; attempted—a transition proven unsatisfiable is dropped), **type-size** (maximum AST depth of a generated label), and **probability-indeterminate** p. On each call, *gentype* returns, with probability p, a predicate type whose subtype relations are indeterminate by construction, and otherwise a type it can prove inhabited. Two structural restrictions follow from Algorithm 4.1 and were noted by a reviewer of the earlier version: the initial simple path forces a connected structure with every state of out-degree ≥ 1, and the only accepting state is q_{n-1}, the far end of that path. Both are instances of Blind spot 1 of Section 4.2.

One might hope to escape this restriction by combining several `path-seeded` outputs with the Boolean operators of Section 2 (intersection, union, exclusive-or) to synthesize automata with richer topology. We expect this to fail for the same reason random RTE syntax trees did, above: each `path-seeded` sample guarantees its own q_0-to-accepting path by construction, but two independently-generated paths have no reason to survive an intersection, so combining automata this way risks the same absorbing-operator collapse

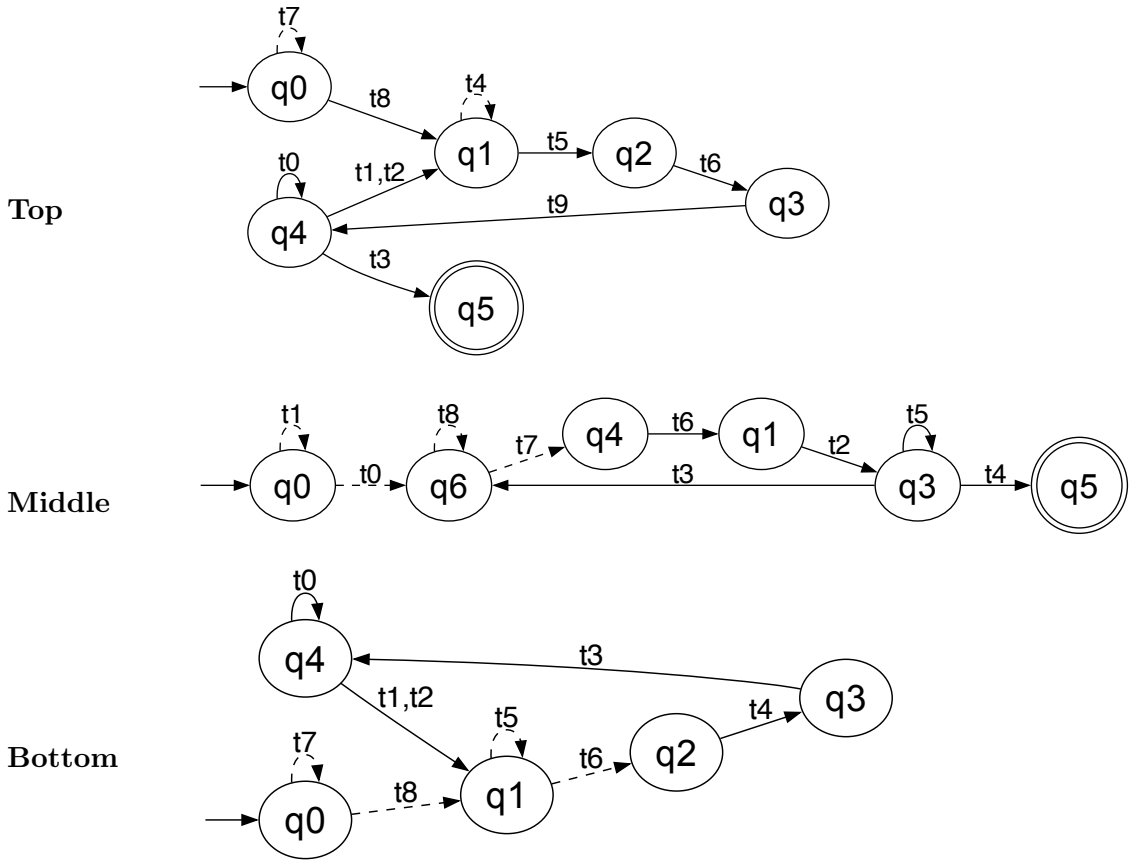


FIGURE 6. (Top) σCDFA generated by `path-seeded`, (Middle) generated from the extracted RTE, (Bottom) difference XOR of Top and Middle. Labels $t1, t2, \ldots$ among different σCDFAs are unrelated.

Koechlin and Rotondo [KR21] describe, one level up—over automata instead of syntax trees. Concatenation would not have this problem, since concatenating two non-empty languages is always non-empty, but chaining `path-seeded` segments end to end would only produce a longer instance of the same single-path topology, not more branching. Section 4.10 instead compares `path-seeded` directly against six structure-first generators that already exist.

4.9. **Results of random generation.** We generated 5040 σCDFAs, 2–49 states ($\mu = 23.03$, $\sigma = 14.49$), 2–140 transitions ($\mu = 45.20$), of which 0–78 were indeterminate ($\mu = 13.86$). Figure 6 shows one sample, its extracted-RTE rebuild, and their (empty) difference automaton—language equivalence verified despite indeterminate transitions in both operands.

Of the samples, 1246 (24.60%) had a provably inhabited language and 3796 (74.95%) were indeterminate. On pairs (A, B) we measured $\mathbb{P}[\mathcal{L}\{A\} \subseteq \mathcal{L}\{B\}] = 23.65\%$ (indeterminate 61.19%) and $\mathbb{P}[\mathcal{L}\{A\} \cap \mathcal{L}\{B\} = \emptyset] = 50.46\%$ (indeterminate 42.94%). Figure 7 plots these against state count. The headline is that for roughly 74.95% of generated σCDFAs habitation is indeterminate and for 40–50% of pairs intersection cannot be decided: the semi-Boolean

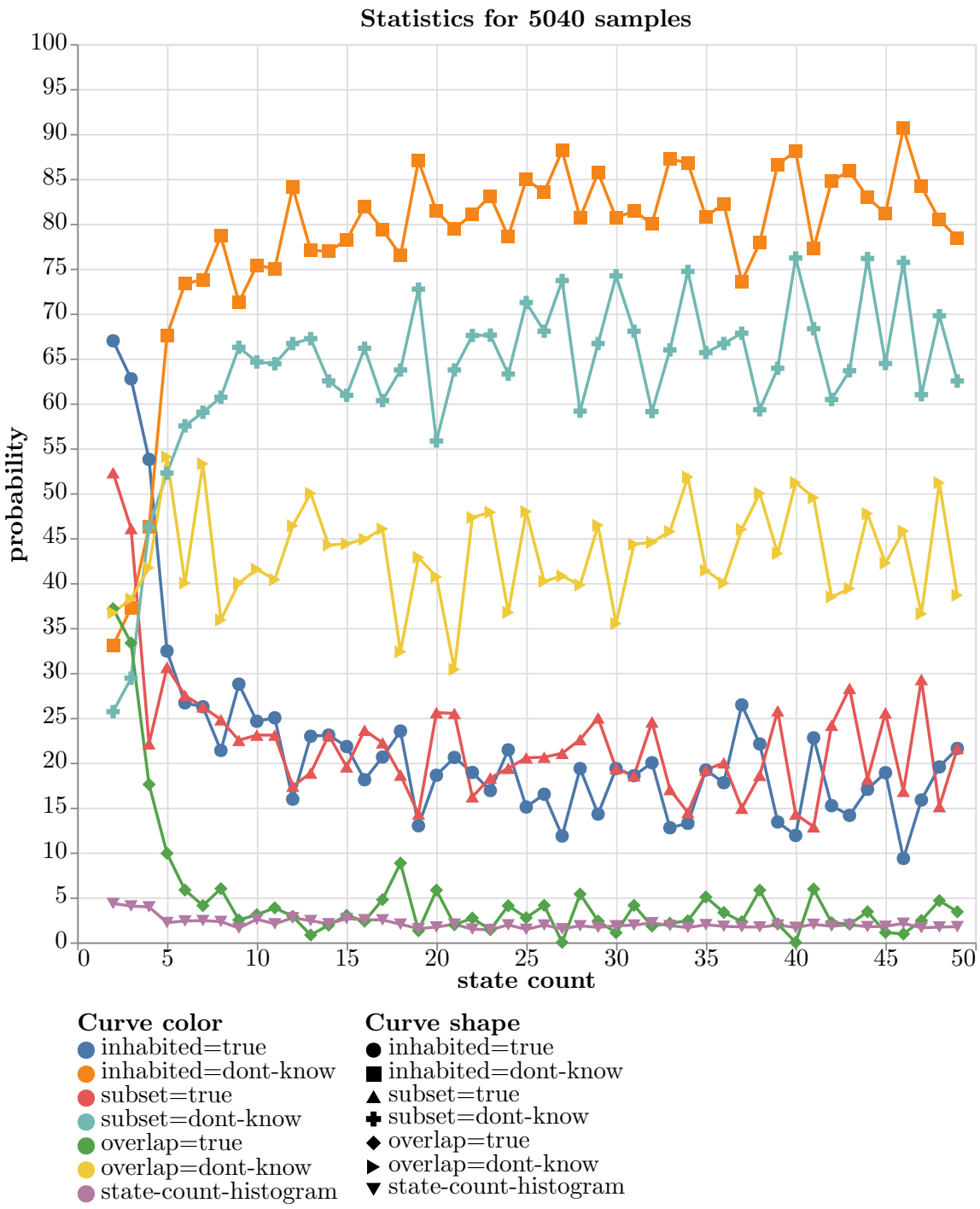
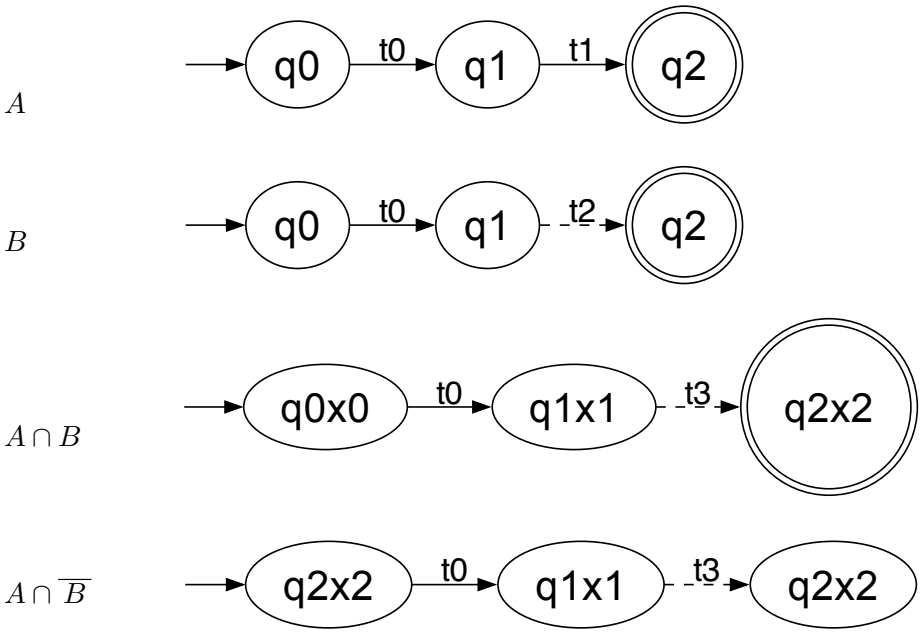


FIGURE 7. Generation summary of 5040 σCDFAs: probabilities of habitation, overlap, and subsetness as a function of state count. The purple, mostly flat curve (state-count histogram), running between 0 and 3%, gives the percentage of total samples (y-axis) at each state count (x-axis); its flatness shows a roughly even sampling of σCDFAs by state count.



t_0 = `:sigma`

t_1 = `(or (and (not (member 1 2 3)) Long) Byte Short)`

t_2 = `(satisfies positive?)`

t_3 =
```
(or (and (not (member 1 2 3)) (satisfies positive?)  Long)
        (and (satisfies positive?)  Byte)
        (and (satisfies positive?)  Short))
```

FIGURE 8. σCDFAs A and B, with $A \cap B$ and $A \cap \overline{B}$. The habitation of $\mathcal{L}\{A \cap B\}$ is indeterminate (its only accepting computation uses t_3, which is indeterminate), while $\mathcal{L}\{A \cap \overline{B}\}$ is empty (no accepting states). The t_i are shown in Clojure s-expression [Dai15] syntax.

third value is not a corner case in this corpus but the common case, which is what makes Theorem 3.3's explicit INDETERMINATE verdict necessary rather than cosmetic.

Figure 8 is one instructive pair: $\mathcal{L}\{A \cap B\}$ has indeterminate habitation (its only accepting computation uses the indeterminate label t_3), yet $\mathcal{L}\{A\} \subseteq \mathcal{L}\{B\}$ is *provable* because $A \cap \overline{B}$ has no accepting state. The raw data and analysis code are public: `github.com/jimka2001/clojure-rte/tree/master/resources/statistics`.

4.10. `path-seeded` **against structure-first generation.** A reviewer of the earlier version raised exactly the structural restriction of Algorithm 4.1 noted above: seeding every σCDFA with an initial simple path excludes a large class of possible σCDFAs. Table 2 puts `path-seeded` next to the six structure-first generators of the companion talks [NZ25a, NZ25b] (tree-split, in three pivot variants; a Flajolet-style random binary-search-tree skeleton;

TABLE 2. Minimized σCDFA state count by generator. n is sample size; μ, σ are the mean and standard deviation of the minimized state count; the last column is the percentage of samples with at least 10 states after minimization, the same statistic used as the headline number in [NZ25a].

generator	n	μ	σ	≥ 10 states
`path-seeded`	5069	22.9	14.5	76.4%
tree-split-linear	410	5.0	8.8	9.0%
tree-split-gauss	410	6.7	35.7	10.2%
tree-split-inv-gauss	410	5.0	10.6	9.5%
Flajolet	410	7.3	20.6	17.1%
root-averse	412	101.9	288.3	76.9%
comb	405	4.7	16.0	3.2%

root-averse, the same skeleton with the annihilating `And` operator's probability graded by depth—rare near the root, common near the leaves, so an accidental disjointness collapses only a small subtree rather than the whole expression, a refinement suggested by an audience member when this generator was first presented; and `comb`, a deliberately adversarial worst case), on the same metric the talks themselves used—minimized state count, and the percentage of samples with at least 10 states. No new generation runs were needed: every row is aggregated directly from the CSV output already on disk from that talk preparation.

`path-seeded`'s 76.4% is far closer to root-averse's 76.9% than to any of the other four structure-first generators, all of which stay under 18%—and this is not a coincidence between two unrelated numbers: root-averse is the one talk generator built specifically to avoid the same collapse-to-triviality failure mode `path-seeded`'s own construction sidesteps by a different means (Section 4.8), so the two landing together is exactly what the underlying mechanism predicts. root-averse's original sample was a fifth the size of the other talk generators' ($n = 194$); augmented here to $n = 412$ (all samples appended, none re-run, via the same locked-append mechanism the original data collection used), the headline percentage moved only from 76.3% to 76.9%—the finding was not an artifact of a small sample. This table, on its own, answers only the size half of the reviewer's observation: `path-seeded` does not produce the small, mostly-trivial automata that plain random RTE-tree generation did. It does not by itself answer the structural half—whether `path-seeded`'s single-path topology still under-samples some class of interesting σCDFAs—which Section 4.8 addresses directly, by argument rather than by data: combining `path-seeded` outputs to escape that topology risks the identical collapse this section's own comparison is about.

5. Conclusion and Perspectives

The central theme of this paper has been the interaction between two layers of reasoning. At the type level, subtype and inhabitation questions may yield the answer `dont-know`. At the automata level, those same type designators serve as transition labels. The main result of the paper is that the resulting three-valued vacuity question nevertheless admits a precise automata-theoretic treatment: Theorem 3.3 reduces it to a shortest-path problem while preserving witness generation. The applications to differential testing and dead-code detection illustrate two settings in which this reduction becomes useful in practice.

In this paper, we have examined how classical finite automata theory can be generalized to symbolic automata over infinite alphabets, particularly when semi-Boolean subtype predicates are introduced. While many familiar constructions still apply, key divergences emerge—most notably in the determinism and decidability of habitation checks.

Our contributions include a new automata-based proof of the Brzozowski derivative of the complement of a regular tree expression, an algorithm for verifying habitation in a subclass of symbolic automata, an engineering-based method for generating randomized symbolic automata, and a property-based testing flow tailored to this symbolic setting. These tools not only help validate theoretical properties but also support the development of practical software libraries for symbolic automata.

Despite our advances, challenges remain. Habitation remains undecidable in general, and the reliability of our testing approach depends on heuristics and biases that may not catch all errors. Nevertheless, the techniques introduced here provide a foundation for further research—both in exploring the boundaries of symbolic automata and in refining related testing methodologies.

A related, more specific limitation concerns *redundant* runtime type checks. Given a σCDFA state with several outgoing transitions, naively deciding which transition a symbol takes, one transition at a time, can call the same underlying predicate more than once. The canonical hard case, unresolved by simple clause reordering, dates to Newton and Verna [NV18b]: two `typecase` clauses testing `unsigned-byte` and `bignum`, together equivalent to their exclusive-or, so that reordering can never avoid checking both predicates on some inputs. Our three implementations address this differently. Scala's `xymbolyco` builds a lazily-expanded decision tree, evaluating each underlying predicate at most once per query and reusing the expansion across repeated queries on the same state. Clojure's `genus` builds a hash-consed reduced ordered binary decision diagram, sharing structurally identical sub-decisions across the whole automaton, in the same lineage as Newton and Verna [NV18b]'s original technique, and further exploits the stability of Clojure's own core library: as discussed in Section 2.5, many built-in predicates in idiomatic use are automatically rewritten to the hosted-type designator they secretly compute [NZ25a]. Python's implementation builds the full decision tree eagerly, before any query is made. These are three different points on the lazy/eager and shared/unshared design space, not the same idea implemented three times.

We fully acknowledge that none of these optimizations has been performance tested, only correctness tested. Meaningful performance testing requires real-world examples, of which we have none, as our system is a research experiment rather than an industrial application. Laboratory-style performance tests, however, could certainly be conducted using the various RTE-tree and σCDFA generators already implemented in this project (Section 4.8). Absent such measurement, the case for the optimization is strongest when a transition's designator wraps an expensive predicate ($Sat(f)$, Section 2.3): avoiding repeated calls to an arbitrarily costly host function is a saving regardless of bookkeeping cost. The case is weakest, and may even be negative, when every candidate designator is a cheap hosted-type check, where the decision-structure machinery itself could plausibly cost more than the redundant checks it avoids. Which case dominates in practice is an open, and eminently measurable, question.

A second, easily stated improvement concerns `minimize` (Section 3.1). All three implementations currently keep two states in the same class only when their combined outgoing labels simplify to identical designators; they never ask whether two differently simplified designators denote the same type. Newton and Verna [NV19] already observed this limitation for the original Common Lisp version of the algorithm: a state whose transitions into one

class are labeled T' and T'' is indistinguishable from a state with a single transition into that class labeled $T' \cup T''$, yet was not merged with it. Our implementations have since combined all labels leading into the same class before comparing, which resolves that example, but only when the combined label simplifies to the same designator. A stronger refinement step would also keep together two states that reach the same destination classes when, for each such class, the exclusive-or of their combined labels is provably empty. Merging only on that certain answer, never on `dont-know`, preserves soundness. The stronger step never finds fewer merges and sometimes finds more, at the price of extra calls to the semi-Boolean procedure, pairwise among the states of a class. Whether the smaller automata repay that cost is, like the question above, measurable but not yet measured.

Future work will include expanding our testing framework to be more theoretically sound, and contrasting with the current engineering approach. Kleene star already underlies the Brzozowski construction central to this paper; whether the shortest-path reduction of Theorem 3.3 can be recast as an instance of the algebraic path problem over a suitably chosen semiring—a framing that would generalize cleanly to graded or probabilistic indeterminacy (Section 3.3)—is a natural question we have not pursued.

Constructing a tree structure randomly (as in Section 4.8) is equivalent to randomly selecting such a structure from the set of all possible structures. We wish this selection to be *uniform* in some sense, but deciding what exactly that means is difficult. Naively, one might like to make a selection under a constraint such as maximum-depth, such that any tree of depth limited by that maximum is equally likely. As Koechlin and Rotondo [KR21] explain, when there are semantics associated to structure, we find that uniformity does not equate to uniformity in semantics. In our case, uniform selection of RTE AST does not correspond to uniform selection of language of the associated automaton. Koechlin and Rotondo suggest a biased structure selection, called *BST-like*, which avoids the problem caused by the annihilators ($A \cap \emptyset = \emptyset$ and $A \cup \Sigma = \Sigma$). As part of expanding our testing framework, we plan to adapt the BST-like construction to our RTE generation to determine whether the method improves our failed approach. Preliminary results seem promising, but are not yet ready to publish.

The engineering approach we are currently using is based on parameterizing values which makes the approach all but impossible to analyze theoretically. We cannot know how uniform our σCDFA selection is nor whether the selection is surjective. Adapting the Koechlin approach promises to make this analysis tractable. Additionally, the Koechlin approach is likely to have applications in property based testing far beyond the mere generation of σCDFA.

6. Acknowledgments

Thanks to Ghiles Ziat `ghiles.ziat@epita.fr` for helpful discussions while developing this paper. Thanks to Antoine Genitrini `antoine.genitrini@lip6.fr` for advice on random generation of tree structures.

An earlier version of this paper, submitted to VMCAI 2026, received proofreading and editorial suggestions from ChatGPT; that chat is public: `https://chatgpt.com/share/687e3798-22e0-800d-9d34-e4a27a2ecd32`.

The present revision was prepared with the assistance of Claude (Anthropic), which drafted some passages and examples at the author's request, checked claims by running code against the implementations discussed, suggested corrections, and helped with formatting

and bibliography. All text was reviewed, edited, and approved by the author, who takes full responsibility for the content of this paper.

Microsoft Copilot was used as a simulated reviewer, identifying places where readers might lose track of the paper's overall narrative, where terminology from type theory and automata theory would benefit from translation, and where additional roadmap sentences, section transitions, and explanatory signposts could improve readability. Particular attention was given to reducing cognitive load for readers approaching the paper from only one of its two disciplinary perspectives and for readers whose primary language is not English. Its contributions were editorial rather than technical.

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. The author declares no competing interests.

References

[Ans94] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[BHM+15] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi:10.1109/TSE.2014.2372785.

[BHM+19] Sabine Broda, Markus Holzer, Eva Maia, Nelma Moreira, and Rogério Reis. A mesh of automata. *Information and Computation*, 265:94–111, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0890540119300033, doi:10.1016/j.ic.2019.01.003.

[Brz64] Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, October 1964. URL: http://doi.acm.org/10.1145/321239.321249, doi:10.1145/321239.321249.

[Cas16] Giuseppe Castagna. Covariance and Contravariance: a fresh look at an old issue. Technical report, CNRS, 2016. URL: https://arxiv.org/pdf/1809.01427.pdf.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/351240.351266.

[Clo26] Clozure Associates. Clozure CL, 2026. Version 1.13. URL: https://ccl.clozure.com/.

[Dai15] Hong-Yi Dai. The mysteries of lisp – i: The way to s-expression lisp, 2015. URL: https://arxiv.org/abs/1505.07375, arXiv:1505.07375.

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[DV13] Loris D'Antoni and Margus Veanes. Equivalence of extended symbolic finite transducers. In *Computer Aided Verification, 25th International Conference (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, pages 624–639. Springer, 2013. doi:10.1007/978-3-642-39799-8_41.

[DV16] Loris D'Antoni and Margus Veanes. Minimization of symbolic tree automata. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2016)*, pages 873–882. ACM, 2016. URL: https://dl.acm.org/doi/10.1145/2933575.2933578, doi:10.1145/2933575.2933578.

[DV17a] Loris D'Antoni and Margus Veanes. Forward bisimulations for nondeterministic symbolic finite automata. In *Tools and Algorithms for the Construction and Analysis of Systems, 23rd International Conference (TACAS 2017)*, volume 10205 of *Lecture Notes in Computer Science*, pages 518–534. Springer, 2017. doi:10.1007/978-3-662-54577-5_30.

[DV17b] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification, 29th International Conference (CAV'17)*. Springer, July 2017. URL: https://www.microsoft.com/en-us/research/publication/power-symbolic-automata-transducers-invited-tutorial/.

[ECL26] ECL developers. ECL: Embeddable Common-Lisp, 2026. Version 26.5.5. URL: https://ecl.common-lisp.dev/.

[FB97] George Fink and Matt Bishop. Property-based testing: A new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, July 1997. doi:10.1145/263244.263267.

[FCB08] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4):19:1–19:64, September 2008. URL: http://doi.acm.org/10.1145/1391289.1391293, doi:10.1145/1391289.1391293.

[FFZ20] Dana Fisman, Hadar Frenkel, and Sandra Zilles. On the complexity of symbolic finite-state automata, 2020. URL: https://arxiv.org/abs/2011.05389, arXiv:2011.05389, doi:10.48550/arXiv.2011.05389.

[GJS+14] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[gra14] Seqexp: regular expressions for sequences, 2014. URL: https://github.com/cgrand/seqexp.

[Gri16] Radu Grigore. Java generics are turing complete. *CoRR*, abs/1605.05274, 2016. URL: http://arxiv.org/abs/1605.05274, arXiv:1605.05274.

[Hai10] Bruno Haible. Gnu clisp, 2010. URL: https://clisp.sourceforge.io.

[Hei22] Mikko Heikkilä. Malli, Metosin, 2022. URL: https://github.com/metosin/malli.

[Hic08] Rich Hickey. The Clojure Programming Language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.

[Hic20] Rich Hickey. A History of Clojure. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. doi:10.1145/3386321.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Hop71] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[KP07] Andrew Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*, January 2007. URL: https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/.

[KR21] Florent Koechlin and Pablo Rotondo. Absorbing Patterns in BST-Like Expression-Trees. In Markus Bläser and Benjamin Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science (STACS 2021)*, volume 187 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:15, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2021.48, doi:10.4230/LIPIcs.STACS.2021.48.

[KT14] Matthias Keil and Peter Thiemann. Symbolic Solving of Extended Regular Expression Inequalities. In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*, volume 29 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 175–186, Dagstuhl, Germany, 2014. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSTTCS.2014.175, doi:10.4230/LIPIcs.FSTTCS.2014.175.

[LS23] Christof Löding and Max Philip Stachon. On minimization and learning of deterministic ω-automata in the presence of don't care words. *Fundamenta Informaticae*, 189(1), 2023. URL: https://arxiv.org/abs/2211.08787, arXiv:2211.08787, doi:10.46298/fi.10322.

[MDS11] Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: A functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 189–195, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2034773.2034801.

[Mil22] Alex Miller. Spec Guide, 2022. URL: https://clojure.org/guides/spec.

[Moo56] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[New15] William H. Newman. Steel Bank Common Lisp User Manual, 2015. URL: http://www.sbcl.org.

[New18] Jim Newton. *Representing and Computing with Types in Dynamically Typed Languages*. PhD thesis, Sorbonne University, November 2018.

[New23] Jim Newton. An elegant and fast algorithm for paritioning types. In *European Lisp Symposium*, Amsterdam, Netherlands, April 2023.

[New24a] Jim Newton. Regular Type Expressions for Clojure, 2024. URL: `github.com/jimka2001/clojure-rte`.

[New24b] Jim Newton. Regular Type Expressions for Common Lisp, 2024. URL: `github.com/jimka2001/cl-rte`.

[New24c] Jim Newton. Regular Type Expressions for Python, 2024. URL: `github.com/jimka2001/python-rte`.

[New24d] Jim Newton. Regular Type Expressions for Scala, 2024. URL: `github.com/jimka2001/scala-rte`.

[New25a] Jim Newton. Recognizing Regular Patterns in Mixed-Type Sequences using Symbolic Finite Automata. Presentation, reClojure 2025, 2025. Non-refereed conference talk. URL: `github.com/jimka2001/clojure-rte/tree/master/doc/reClojure.2025`.

[New25b] Jim Newton. Type-checking heterogeneous sequences in a simple embeddable type system. In Esra Erdem and Germán Vidal, editors, *Practical Aspects of Declarative Languages*, pages 18–34, Cham, 2025. Springer Nature Switzerland.

[NP21] Jim Newton and Adrien Pommellet. A portable, simple, embeddable type system. In *European Lisp Symposium*, Online, Everywhere, May 2021.

[NV18a] Jim Newton and Didier Verna. Recognizing hetergeneous sequences by rational type expression. In *Proceedings of the Meta'18: Workshop on Meta-Programming Techniques and Reflection*, Boston, MA USA, November 2018.

[NV18b] Jim Newton and Didier Verna. Strategies for typecase optimization. In *European Lisp Symposium*, Marbella, Spain, April 2018.

[NV19] Jim Newton and Didier Verna. Finite automata theory based optimization of conditional variable binding. In *European Lisp Symposium*, Genova, Italy, April 2019.

[NVC17] Jim Newton, Didier Verna, and Maximilien Colange. Programmatic Manipulation of Common Lisp Type Specifiers. In *European Lisp Symposium*, Brussels, Belgium, April 2017.

[NZ25a] Jim Newton and Ghiles Ziat. Balanced Sampling and Useful Random Tree Generation. Presentation, Clojure/conj 2025, November 2025. Non-refereed conference talk. URL: `github.com/jimka2001/clojure-rte/tree/master/doc/clojure.conj/2025`.

[NZ25b] Jim Newton and Ghiles Ziat. Balanced Sampling as a Tool for Useful PBT Random Tree Generation. Presentation, Scala.io 2025, Nantes, France, November 2025. Non-refereed conference talk. URL: `github.com/jimka2001/scala-rte/tree/master/doc/scala.io/2025`.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala language specification, 2004.

[ORT09] Scott Owens, John Reppy, and Aaron Turon. Regular-expression Derivatives Re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Sigplan Notices - SIGPLAN*, volume 40, pages 41–57, 10 2005. `doi:10.1145/1103845.1094815`.

[Pfl73] C. P. Pfleeger. State reduction in incompletely specified finite-state machines. *IEEE Transactions on Computers*, 22(12):1099–1102, 1973. `doi:10.1109/T-C.1973.223655`.

[Sak09] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, USA, 2009.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002. `doi:10.1145/514188.514190`.

[vRD11] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011.