Vacuity Checks in Symbolic Finite Automata Using Semi-Boolean Predicates

Jim Newton[0000-0002-1595-8655]

EPITA Research Lab, 94270 Le Kremlin Becêtre, FRANCE 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/dont-know), these operations can become undecidable and nondeterministic. This divergence introduces subtle but critical challenges for both the theory and practice of symbolic automata.

In this paper, we explore this divergence in depth, focusing on the failure of determinism in vacuity and habitation checks. To support our theoretical results, we revisit and generalize several foundational constructions from classical automata theory, adapting them to accommodate challenges of symbolic setting. We also describe a property-based testing strategy for symbolic automata libraries, implemented in Clojure, that uses randomized automaton generation and difference checking to detect semantic inconsistencies. In particular, we present an algorithm that can often distinguish habitation from vacuity efficiently, even in undecidable settings. These findings have practical implications for the reliability of symbolic automata tools and the correctness of automata-based language processing.

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.

However, when symbolic automata are introduced—especially over infinite alphabets with semi-Boolean (true/false/dont–know) subtype predicates—new challenges emerge: undecidabile 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, 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 the central contribution 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 construct 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.

Building on this theoretical groundwork, we [28] have implemented symbolic finite automata libraries [25, 27, 26, 24] in several programming languages: Common Lisp [2], Scala [32, 33], Python [35], and Clojure [14]. This paper focuses on the Clojure-based implementation, and further expounds upon how such a library can be tested for correctness. We use a 5-step, property-based testing flow that includes both previously established and newly developed components:

1. (New) Construct random finite automaton: Section 4.2.
2. Extract a regular expression from it: [36, Sec 2.4.2].
3. Reconstruct a finite automaton from the regular expression: Section 2.2.
4. XOR two finite automata producing a difference-automaton: Newton [31].
5. (New) Assert language vacuity of the difference-automaton: Section 3.

If the flow finds the language of the *difference-automaton* unexpectedly inhabited, then we know there is a bug in our software library. In this work we elaborate on steps 1, 3, and 5 of the flow. Step 2 follows existing state of the art, and is left for future refinement. Step 4 is previous work. While step 3 has been addressed in previous work, we revisit and clarify key theoretical aspects here.

1.1 Our Contribution

1. A finite automata-based proof of the Brzozowski derivative of the complement of an RTE.
2. An algorithm for proving habitation of some σCDFAs.
3. An algorithm for generating randomized symbolic finite automata.
4. A property-based testing flow to partially verify σCDFA library.

1.2 Previous Work

Brzozowski [4] introduced the derivative in 1964. Owens *et al.* [34] *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 [22], Newton used deterministic finite automata (DFA) and RTEs to recognize heterogeneous Common Lisp [2] sequences based on type patterns. Newton and Verna [30] addressed certain meta programming aspects of RTEs. Newton [28] generalized RTEs to a wider range of programming languages.

D'Antoni and Veanes [7] and Newton [28] 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.

Grigore [12], Kennedy and Pierce [18], discuss subtype decidability in Java [11], Scala [32, 33], 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 2.1 we discuss types as sets. Newton [29] introduced SETS to *designate* types with set semantics in Python [35], Clojure [14], and Scala [32], inspired by the Common Lisp [2] type system. Frisch [10] *et al.* discuss set-based type semantics in non-dynamic languages, arguing against 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 [2, Section TYPEP] explicitly forbids this runtime check.

Clojure Spec [21, 15, 14], `metosin` [13], and `seqexp` [1] support forms of type pattern recognition. Spec is based on finite automata theory at all or rather on NFA (non-deterministic finite automata) work by Might *et al.* [20], and thus leading to exponential complexity because of backtracking. As far as we can tell, `seqexp` does not use a finite automata approach.

2 Symbolic Finite Automata

In this section we outline the theoretical and practical construction of symbolic finite automata (σFA). We base our description heavily on intuition from classical finite automata theory. The reader already intimately familiar with the classical theory might want to skim this section, concentrating primarily on Definitions 2 and 3, and Proposition 2.

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

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 $\Sigma \subset \Sigma$ and $\emptyset \subset \Sigma$ are types.

Definition 2 (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 is to later define σCDFA in Definition 8.

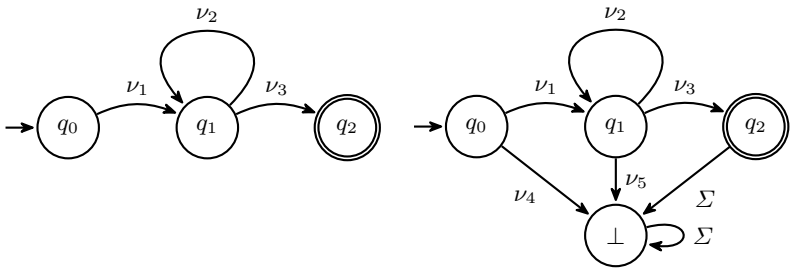


Fig. 1: Symbolic Finite Automata σFA: (left) incomplete, (right) complete assuming $\nu_4 = \Sigma \setminus \nu_1$, and $\nu_5 = \Sigma \setminus (\nu_2 \cup \nu_3)$. The additional state, $\perp$, called the sink state, is explained in Proposition 1.

Definition 3 (un/satisfiable transition). *A transition $\textcircled{q} \xrightarrow{\nu} \textcircled{r}$ is called* unsatisfiable *if $[\![\nu]\!] = \emptyset$ even if $\nu \neq \emptyset \in \Upsilon$; else the transition is called* satisfiable*.*

Note that it is possible that $[\![\nu]\!] = \emptyset \subset \Sigma$ without $\nu = \emptyset \in \Upsilon$; *e.g.*, $\nu \cap \overline{\nu} \in \Upsilon$ is a symbol in the type space distinct from the symbol $\emptyset \in \Upsilon$. However, $[\![\nu \cap \overline{\nu}]\!] = [\![\nu]\!] \cap [\![\overline{\nu}]\!] = \emptyset \subset \Sigma$.

Definition 4 (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 5 (partition). *A* partition *of set S is a set of mutually disjoint, possibly empty subsets of S whose union is S.*

Definition 6 (disjoint symbolic decomposition). *Let (Σ, Υ) be a complemented type space. If $S = \{\mu_1, \mu_2, \ldots, \mu_m\} \subset \Upsilon$, then the set $D = \{\nu_1, \nu_2, \ldots, \nu_n\} \subset \Upsilon$ is called a* disjoint symbolic decomposition *of S if*

1. $1 \leq i \leq n$, $1 \leq j \leq m \implies (\llbracket \nu_i \rrbracket \subset \llbracket \mu_j \rrbracket) \vee \left(\llbracket \nu_i \rrbracket \subset \overline{\llbracket \mu_j \rrbracket} \right)$.
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 $S \subset \Sigma$.

Definition 7 (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*

$$DD(\nu, \mu) = \left\{ \nu \cap \mu, \overline{\nu} \cap \mu, \nu \cap \overline{\mu}, \overline{\nu} \cap \overline{\mu} \right\} \qquad \subset \Upsilon \qquad (1)$$

$$DD(\nu_1, \nu_2, \ldots, \nu_n) = \bigcup_{\nu \in DD(\nu_2, \ldots, \nu_n)} DD(\nu_1, \nu) \cup DD\left(\overline{\nu_1}, \nu\right) \qquad \subset \Upsilon \qquad (2)$$

$$SP(\nu_1, \nu_2, \ldots, \nu_n) = \{\llbracket \nu \rrbracket | \nu \in DD(\nu_1, \nu_2, \ldots, \nu_n)\} \qquad \subset 2^{\Sigma} \qquad (3)$$

The default decomposition (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 $\llbracket \nu \rrbracket = \llbracket \mu \rrbracket$, $\llbracket \nu \rrbracket \subset \llbracket \mu \rrbracket$, or$\llbracket \nu \rrbracket \subset \overline{\llbracket \mu \rrbracket}$.

Definition 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., $\forall q \in Q$, $\bigcup_{\nu \in \xi_A(q)} \llbracket \nu \rrbracket = \Sigma$.*
2. deterministic*, σDFA, if the elements of $\xi_A(q)$ are mutually disjoint; i.e., if $\forall q \in Q, \forall \nu_1, \nu_2 \in \xi_A(q), \nu_1 \neq \nu_2 \implies \llbracket \nu_1 \cap \nu_2 \rrbracket = \emptyset$.*

We denote a complete, deterministic σFA as $\sigma CDFA$.

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

Proposition 1 (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: $\textcircled{q} \xrightarrow{\overline{\bigcap_{\nu \in \xi_A(q)} \nu}} \textcircled{\perp}$. More explicitly: $A' = (\Sigma, \Upsilon, Q', q_0, F, T')$ is complete provided*

$$Q' = Q \cup \left\{ \textcircled{\perp} \right\}$$

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

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 9 (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 10 (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 11 (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.*

Proposition 2 (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, $T_0, T_1, T_2, \ldots, T_m$:

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

Now let $N' = (\Sigma, \Upsilon, Q, q_0, F, T')$, and let $\Sigma' = \bigcap_{q \in Q} \xi_{N'}(q)$. Lo and behold, (Σ', Q, q_0, F, T') is a classical FA. Indeed for any two distinct transitions (p, ν, q_1) and (p, μ, q_2), sharing the same origin, then their labels, ν and μ, either designate the exact same type, $[\![\nu]\!] = [\![\mu]\!]$ or disjoint types, $[\![\nu]\!] \cap [\![\mu]\!] = \emptyset$. Thus the FA can be determinized through classical techniques [16].

Definition 12 (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 3. *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 successful computations, then every unsuccessful 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 a successful computation in A, then it ends in a state in $q \in F$, thus does not end in a state in $Q \setminus F$; hence t is not successful in $\overline{A}$. Likewise if t is an unsuccessful computation in A, then it ends in a state $q \in Q \setminus F$; hence t is successful in $\overline{A}$. □

Definition 13 (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 *[3] of A at q_1.*

Proposition 4. *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 a successful computation accepting $(s_1, s_2, \ldots, s_n)$. *I.e.*, $s_1 \in [\![\nu_1]\!]$, $s_1 \in [\![\nu_1]\!]$, $\ldots$, $s_n \in [\![\nu_n]\!]$. Thus $(s_2, \ldots, s_n)$ is accepted by the computation $(\nu_2, \ldots, \nu_n)$ on $\overrightarrow{A}_{q_1}$. □

2.1 Rational Type Expressions

Definition 14 (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 $\mathcal{L}\{A\} = [\![r]\!]$. We take Υ to be an extension of the built-in type system of a host

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	$\{(a) \mid a \in \Sigma\}$	$f \,\&\, g$	$[\![f]\!] \cap [\![g]\!]$	$!f$	$\Sigma^* \setminus [\![f]\!] = \overline{[\![f]\!]}$

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

programming language. In [29] and [28], 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, floats etc. This embedding implements a complemented type lattice.

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 characteristic/feature of the SETS embedding is that run-time membership query is always answerable, so we say that there is a Boolean membership predicate. However, the subtype question may sometimes be unanswerable. 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.

As an example where the subtype predicate answers `dont-know`: let f and g be user-defined predicates, $Sat(f) \subset Sat(g)$ returns `dont-know`. Other limitations [29] in the base language may also lead to `dont-know`.

2.2 Brzozowski Derivative Construction of σCDFA

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 [28]. While Brzozowski/Owens [4, 34] defined $\partial_a\, r$, for the letter $a \in \Sigma$, we define $\partial_\nu\, r$ for type designator $\nu \in \Upsilon$.

Definition 15. *Given a type designator $\nu \in \Upsilon$ and an RTE r, a* Brzozowski derivative*, denoted $\partial_\nu\, r$ is any RTE such that*

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

$$\partial_\nu \emptyset = \partial_\nu \varepsilon = \emptyset \quad (4)$$

$$\partial_\nu (r^*) = \partial_\nu r \cdot r^* \quad (5)$$

$$\partial_\nu (r + s) = \partial_\nu r \ + \ \partial_\nu s \quad (6)$$

$$\partial_\nu (r \,\&\, s) = \partial_\nu r \ \& \ \partial_\nu s \quad (7)$$

$$\partial_\nu !r = !\partial_\nu r \quad (8)$$

$$\partial_\nu (r{\cdot}s) = \begin{cases} (\partial_\nu r) \cdot s & \text{if } () \notin [\![r]\!] \\ (\partial_\nu r) \cdot s + \partial_\nu s & \text{if } () \in [\![r]\!] \end{cases} \quad (9)$$

$$\partial_\nu \lfloor \mu \rfloor = \varepsilon \quad \text{if } [\![\nu]\!] \subseteq [\![\mu]\!] \quad (10)$$

$$\partial_\nu \lfloor \mu \rfloor = \emptyset \quad \text{if } [\![\nu]\!] \cap [\![\mu]\!] = \emptyset \quad (11)$$

$$\partial_\nu \lfloor \mu \rfloor \quad \text{otherwise, no rule defined} \quad (12)$$

Fig. 3: Brzozowski Derivative Rules. For Equation (8) see Section 2.5. Variables, ν and μ represent type designators, $[\![\nu]\!], [\![\mu]\!] \subseteq \Sigma$; while r and s represent RTEs.

2.3 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 [23, 28]. MDTD produces a disjoint symbolic decomposition (Definition 6) 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.

Next, construct $q_1, \ldots, q_n$ and associate q_i with $\partial_{\nu_i} r$, computed as in Section 2.4. 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 15. Therefore, we construct the following transitions, with common origin q_0: $(q_0, \nu_1, q_1), (q_0, \nu_2, q_2), \ldots, (q_0, \nu_n, q_3)$. Now repeat the process on newly discovered states and corresponding RTEs, until no new RTEs are discovered as a result of computing the derivative.

2.4 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 [28], we introduced subtype-based rules (10), (11) and (12) which generalize rules from Owens [34].

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

$$\mathcal{L}\{\overrightarrow{A}_q\} = [\![\partial_\nu r]\!], \quad (13)$$

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

Owens [34] and Keil [17] 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 (10) and (11) will be applied.

2.5 Derivative of Complement

Equation (8), $\partial_\nu !r = !\partial_\nu\, r$, is not immediately evident. Directly applying the definitions from above, we see that Equations (14) and (15) are not identical.

$$\begin{aligned}
[\![\partial_\nu !r]\!] &= \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!]\ h :: s \in [\![!r]\!]\} && \text{By Def 15} \\
&= \left\{ s \in \Sigma^* \;\middle|\; \exists h \in [\![\nu]\!]\ h :: s \in \overline{[\![r]\!]} \right\} && \text{By Fig 2} \\
&= \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!]\ h :: s \notin [\![r]\!]\} && (14) \\
[\![!\partial_\nu\, r]\!] &= \overline{[\![\partial_\nu\, r]\!]} && \text{By Fig 2} \\
&= \overline{\{s \in \Sigma^* \mid \exists h \in [\![\nu]\!]\ h :: s \in [\![r]\!]\}} && \text{By Def 15} \\
&= \{s \in \Sigma^* \mid \forall h \in [\![\nu]\!]\ h :: s \notin [\![r]\!]\} && \text{De Morgan} \quad (15)
\end{aligned}$$

In [28], we sketched a proof of Equation (8) which we clarify here. The statement of the proposition critically depends on the claim that any σFA can be determinized (Proposition 2).

Proposition 5. *Given an RTE, r, over alphabet, Σ, whose symbols are $\mu_1, \mu_2, \ldots, \mu_m$ and $(\nu_1, \nu_2, \ldots, \nu_n) \in \Upsilon^n$ designates a partition of Σ such that for each i, j, either $\nu_i \subset \mu_j$ or $\nu_i \subset \overline{\mu_j}$. And given a σCDFA, $A = (\Sigma, \Upsilon, Q, q_0, F, T)$, such that $\mathcal{L}\{A\} = [\![r]\!]$.*

Then $\nu \in \{\nu_1, \nu_2, \ldots, \nu_m\} \implies [\![\partial_\nu !r]\!] = [\![!\partial_\nu\, r]\!]$.

Overview of proof: As illustrated in Figure 4, from A, we construct σCDFAs, B, and C, such that $B = \overrightarrow{A}_{q_1}$, $C = \overline{A}$. We derive D two different ways: $D = \overline{B} = \overrightarrow{C}_{q_1}$, thus showing $[\![\partial_\nu !r]\!] = [\![!\partial_\nu\, r]\!]$.

Proof. 1. Since A is deterministic, and q_0 has outgoing transition $(q_0) \xrightarrow{\nu} (q_1)$, then from Equation (13), we know $\mathcal{L}\{\overrightarrow{A}_{q_1}\} = [\![\partial_\nu\, r]\!]$. We will follow how this transition is transformed in constructing B, C, and D.

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 3). $\mathcal{L}\{B\} = \mathcal{L}\{\overrightarrow{A}_{q_1}\} = [\![\partial_\nu\, r]\!]$.
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 (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** $\overline{\overrightarrow{A}_{q_1}}$**:** Finally, we remark that

$$\overline{B} = (\Sigma, \Upsilon, Q, q_1, Q \setminus F, T) = \overrightarrow{C}_{q_1}\,. \tag{16}$$

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

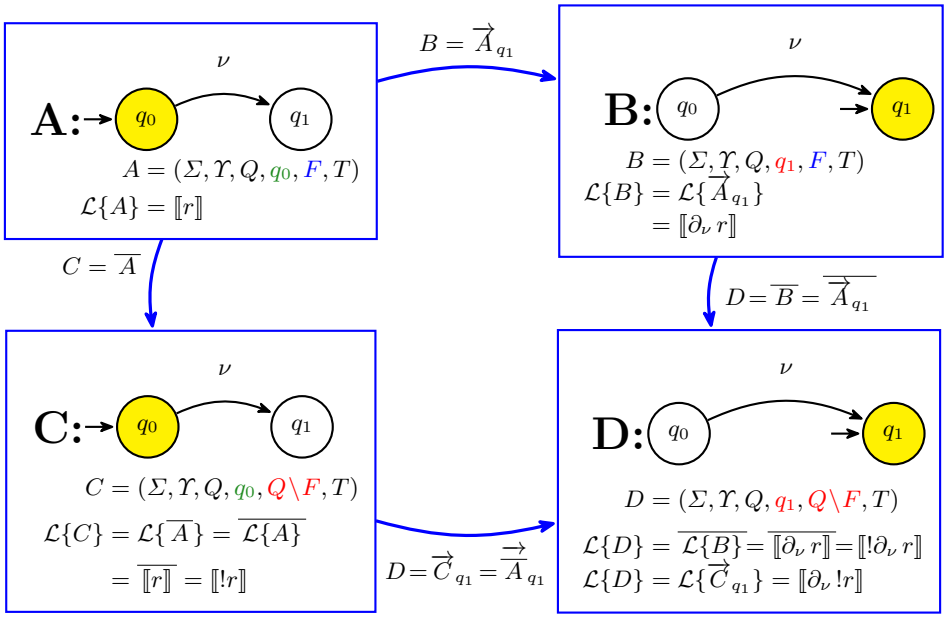


Fig. 4: Transformations of σCDFA: A is transformed to produce B and C. D is constructed by transforming either B or C illustrating that $\overrightarrow{\overline{A}}_{q_1} = \overline{\overrightarrow{A}_{q_1}}$.

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, $[\![\partial_\nu !r]\!] = [\![!\partial_\nu r]\!]$. □

3 Vacuity Checks in a σCDFA

In this section we introduce our procedure for determining habitation of a σCDFA. Habitation checks play an important role in verifying to σCDFAs express the same language as will be see in Section 4. Under which conditions can we verify the habitation or vacuity of the language of a given σCDFA?

There seems to be some surprise among certain experts at the use of the word *habitation*. At the risk of sounding redundant to other readers, a *habitation check* is an attempt to prove a set is inhabited (has a member, is not empty). A constructive habitation check is an attempt find an element. A *vacuity check* can sometimes be implemented as a habitation check with reversed parity of return value, and can sometimes be implemented as a proof that no member exists.

Hopcroft [16] describes one classical approach to non-constructive habitation checks for a finite automaton, A. First, *minimize* the finite automaton, pruning non-accessible states, and non-co-accessible states (states, q, for which $\mathcal{L}\{\overrightarrow{A}_q\} = \emptyset$). Thereafter, $\mathcal{L}\{A_{min}\}$ is empty if and only if there are no accepting

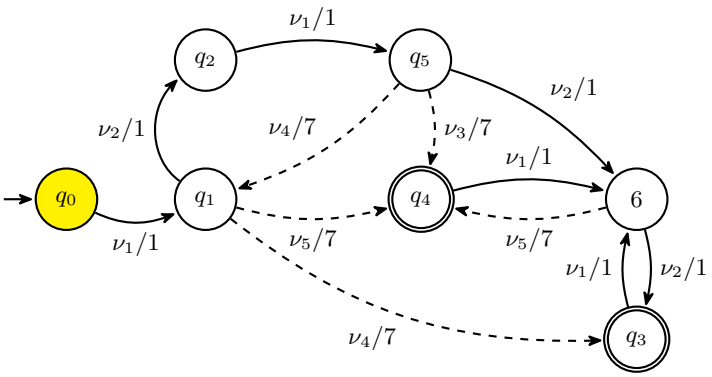


Fig. 5: σCDFA expressed as Dijstra-compatible, weighted graph. $W = 1$ for satisfiable and $W = 7$ for indeterminate transitions.

states. Contrary to classical theory, a σCDFA based on a semi-Boolean subtype procedure cannot be dependably, completely minimized for several reasons. The most obvious reason is that we cannot even dependably determine whether a transition is satisfiable. Thus we cannot dependably identify unreachable states. Nevertheless, our `minimize` function often reduces the number of states, even if it fails to guarantee optimal minimization.

Arguably the `minimize` function is misnamed. There is no good English word which means: make more minimal even if not completely minimal. The name `reduce` or `optimize` would be even more misleading.

Consider the σCDFA in Figure 5. We omit the sink state (Definition 1) and its incoming transitions. We also denote known, satisfiable transitions with solid arrows, and indeterminate transitions with dashed arrows. In particular ν_3, ν_4 and ν_5 are assumed to be indeterminate.

The two accepting states, q_3 and q_4, are reachable over various computations. For example: a computation (ν_1, ν_5) reaches q_4. However, since ν_5 is indeterminate, we do not know if an actual length-2 sequence, $(s_1, s_2) \in \Sigma^*$, exists for which $s_2 \in [\![\nu_5]\!]$. Likewise q_3 is reachable with the computation (ν_1, ν_4), and since ν_4 is indeterminate, we do not know whether such a sequence of objects, $(s_1, s_2) \in \Sigma^*$ exists. Nevertheless, q_3 is also reachable with a computation $(\nu_1, \nu_2, \nu_1, \nu_2, \nu_2)$. Since the types, ν_1 and ν_2, are known to be inhabited, we know there exists at least one sequence in the language of this σCDFA.

The general question is, given a σCDFA, is it inhabited? The answer may be `yes`, `no`, or `dont-know`.

Clearly if there is no accepting state, then the language is vacuous. If there exists an accepting, satisfiable computation (Definition 10), then the language is inhabited. If there are accepting computations, yet every accepting computation is indeterminate, then we have failed to prove the language is vacuous and we have failed to prove it is inhabited. We say the habitation is indeterminate.

One naive way to make this decision is to use a two-pass approach. First, perform a graph search ignoring indeterminate transitions. If an accepting computation is discovered, then we have proven the language is inhabited. Otherwise, perform a second search considering all transitions. If this search discovers an accepting computation then the computation is necessarily indeterminate, and thus the habitation of the language is indeterminate. Assuming there are n states and m total transitions, such a search (ignoring the complexity of deciding habitation) has complexity $O(m + n \log n)$, which we have to perform twice in worst case. The complexity $O(m + n \log n)$ is assuming we are using a `Set` data structure to avoid cycles. If the BFS uses a `HashSet` data structure instead, the complexity is lowered to $O(m + n)$.

Another way to answer the habitation question is based on a shortest path graph search. We often want to know not only whether the language is inhabited, but also we wish to characterize a sequence in the language, *i.e.*, a witness of habitation, and a witness as *simple* as possibly. If we view the σCDFA as a weighted, directed n-vertex graph with origin q_0. The weight of an edge is 1 if the transition is satisfiable. Thus, if there is a non-looping, accepting, satisfiable computation, its weight necessarily $W < n$, as a path of with n edges must repeat a vertex. Because the maximum accepting, satisfiable computation length is n–1, we assign a weight of n to edges corresponding to indeterminate transitions.

With this weighting scheme, if the shortest weighted path search discovers a path with weight, $W \geq n$, then the path is necessarily indeterminate.

Such a computation may actually be unsatisfiable, but by definition, we are unable to judge. The Dijkstra [8] shortest path algorithm can be implemented with complexity $O(n \log n + m)$.

In Figure 5, we assign weights of 1 to ν_1 and ν_2, and weights of 7 to ν_3, ν_4 and ν_5. With this weighting the shortest path from q_0 to q_3 is $(\nu_1, \nu_2, \nu_1, \nu_2, \nu_2)$ and has weight 5, thus is satisfiable (because $5 < 7$). However, the shortest path from q_0 to q_4 is (ν_1, ν_5) and has weight $W = 8$, thus is indeterminate. There are other paths from q_0 to q_4 such as $(\nu_1, \nu_2, \nu_1, \nu_3)$ which has weight $W = 10$. It might very well be that the latter is satisfiable and the former is not. However, again we insist that we cannot distinguish between indeterminate-satisfiable and indeterminate-unsatisfiable.

4 Library Testing

In this section we describe how we use the development of the previous sections to design a test for a software library which manipulates σCDFA. The question "Can we determine the habitation of a σCDFA?" plays an important role in library testing, in particular in verifying the equivalence of two σCDFAs as in Proposition 6.

Proposition 6. *Two σCDFAs, A and B, express the same language if and only if the XOR $(\mathcal{L}\{A\} \oplus \mathcal{L}\{B\})$ is empty.*

Proof.

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

□

We concentrate on testing a particular σCDFA library `clojure-rte` [24] written in Clojure [14]. We use a *property-based testing* (PBT) [9, 5] approach to test some functions in our σCDFA library. Rather than checking hard-coded test cases, we instead describe invariant properties and then test those invariants on randomly generated input, starting with simple structures and advancing to more structurally complex input data.

4.1 Testing Flow

We wish to test the following flow based on random structure generation. The testing should (1) improve our confidence that these functions compute correct values, (2) exhibit that the code actually runs without encountering unexpected exceptions, which is important in dynamic languages with dynamic type systems, and (3) give us some intuition of the overall time-performance of the functions.

1. Generate σCDFA, A, using `gen-dfa` and `minimize`, Figure 6 (Top).
2. Extract its RTE, r, from A, using `dfa-to-rte`.
3. Construct a σCDFA, B, from r, using `rte-to-dfa`, Figure 6 (Middle).
4. Construct $A \oplus B$ automaton using `synchronized-xor`, Figure 6 (Bottom).
5. Check the habitation of $\mathcal{L}\{A\} \oplus \mathcal{L}\{B\} = \mathcal{L}\{A \oplus B\}$ using `dfa-inhabited?`.

If the library code is correct, the languages $\mathcal{L}\{A\}$ and $\mathcal{L}\{B\}$ should be identical and consequently $\mathcal{L}\{A\} \oplus \mathcal{L}\{B\}$ should be empty (Proposition 6), otherwise we have evidence (and a witness) of a bug in one of the functions in our flow.

There are several challenges to this testing approach. (1) The habitation of $\mathcal{L}\{A\}$ may not be known, which will lead to (2) the habitation of $\mathcal{L}\{B\}$ not being known. (3) Given two σCDFAs whose habitation is indeterminate, can we verify that their languages are identical? Additionally (4) given two σCDFAs whose habitation is known, can we verify the equality of their languages, particularly if $A \oplus B$ contains indeterminate transitions?

The high-level functions of interest are.

- `gen-dfa` : Generate a σCDFA with a target number of states and transitions. This generation serves to feed the property-based tester, Section 4.2.
- `minimize` : Simplify a σCDFA using the **DFA Simplification** algorithm presented in [31]— a generalize Hopcroft [16] minimization.

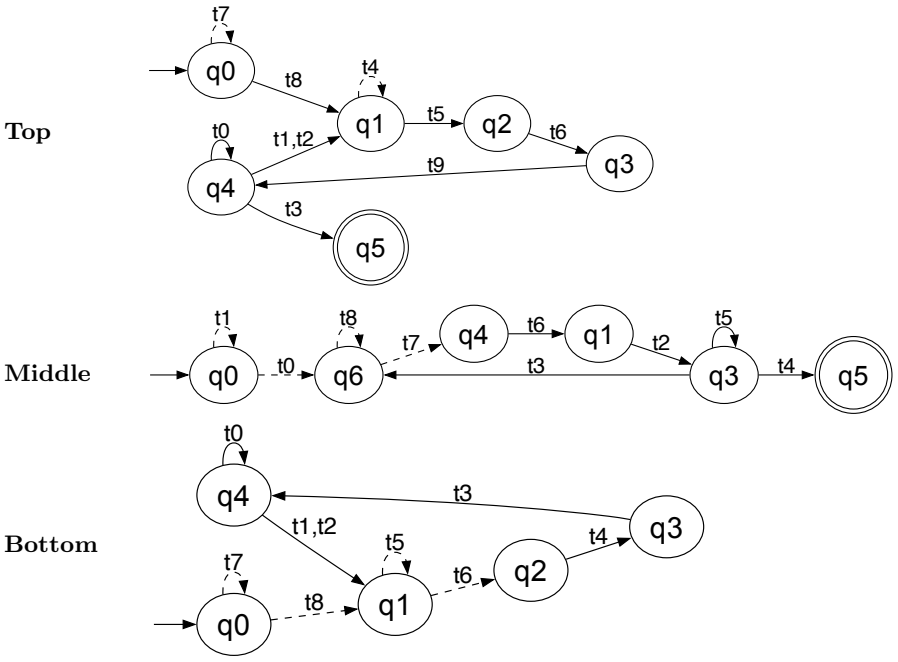


Fig. 6: (Top) σCDFA generated by `gen-dfa`, (Middle) generated from the extracted RTE, (Bottom) difference XOR Top and Middle. Labels $t1, t2, \ldots$ among different σCDFAs are unrelated.

- `dfa-to-rte` : Extract RTE from σCDFA using an unoptimized BMC state elimination method. [36, Sec 2.4.2] Developing a smarter extraction procedure for σCDFA is a matter for future research.
- `rte-to-dfa` : Construct a σCDFA from a given RTE using the Brzozowski derivative summarized in Section 2.2.
- `synchronized-xor` : Given two σCDFAs, A and B, generate a σCDFA having language $\mathcal{L}\{A\} \oplus \mathcal{L}\{B\}$, which is empty if $\mathcal{L}\{A\} = \mathcal{L}\{B\}$ (Proposition 6). Uses an `XOR` algorithm called **Synchronized Cross-Product** [31].
- `dfa-inhabited?` : Habitation check by Dijkstra search. Searches computations spanning σCDFA (Section 3). Returns `true`, if a successful accepting computation is found; `false`, if no successful computation is found; otherwise `:dont-know`, if every accepting computation is indeterminate.

4.2 Randomized σCDFA Construction

There are many possible approaches to generating a random σCDFA. The first approach we tried and have temporarily abandoned (pending future investigation), was to randomly generate an RTE and thereafter generate the σCDFA

Input: n: number of states
Input: m: number of transitions
Input: *gentype*: function to generate a type designator enforcing `type-size` and `probability-indeterminate`.

4.1.1 **begin**
4.1.2 $T \leftarrow []$;
`// Create states, and simply connected transitions`
4.1.3 **for** $i \leftarrow 1 \ldots n$ **do**
4.1.4 $\nu \leftarrow gentype()$;
4.1.5 $T \leftarrow (q_{i-1}, \nu, q_i) :: T$
4.1.6 **for** $i \leftarrow 1 \ldots m - n$ **do**
`// forward, backward, or self-loop transition`
4.1.7 $(p, q) \leftarrow (rand(n), rand(n))$;
`// compute` ν `disjoint with other types having origin` q
4.1.8 $\nu \leftarrow gentype() \setminus \bigcup_{\mu \in \xi_A(q)} \mu$;
4.1.9 $T \leftarrow (q, \nu, p) :: T$

Algorithm 4.1: Compute States and Transitions of Random σCDFA

therefrom. We built the corresponding AST (abstract syntax tree), each internal node selecting randomly among the RTE operations of concatenation, intersection, union, and Kleene-star. What we found, anecdotally, was that the approach generates ASTs, uniform in the syntax space (uniform distribution of RTE operators), but the language thereof was most often either empty or trivial. After applying the minimization function, the σCDFA most often had 1 or 2 states. Koechlin and Rotondo [19] discuss a reason for this; see Section 5.

We now explain the engineering approach for generating random σCDFAs which we settled upon. The `gen-dfa` function takes four parameters which are related to the randomization process. The states and transitions of the σCDFA are constructed as summarized in Algorithm 4.1.

- **num-states** — requested number of states (pre-minimized) in the σCDFA.
- **num-transitions** — requested number of transitions to *attempt* to create. In some cases a transition may be eliminated for being unsatisfiable .
- **type-size** — each transitions is labeled with a randomly selected type designator whose maximum AST depth is limited to this parameter value.
- **probability-indeterminate** — When a type designator is chosen, this is the probability of the type's habitation being indeterminate.

We performed two suites of simulations and we report here some of our randomization results. We generated 4909 σCDFAs ranging from minimum size of 2 states to maximum size of 49 states, with mean size, $\mu = 23.55$ and standard deviation, $\sigma = 14.33$. The number of transitions for these σCDFAs ranged from 2 to 140, with $\mu = 46.22$ and $\sigma = 30.70$. The number of indeterminate transitions ranged from 0 to 78, with $\mu = 14.21$ and $\sigma = 12.25$.

4.3 Results

Figure 6 (Top) shows a σCDFA created by `gen-dfa` containing 6 non-sink states and 10 unique labels. The type designator values of the labels are not shown in the figure as they are machine generated, and not very insightful to the human.

The testing flow generates an RTE (much too large to print on this page), and regenerates a σDFA from that RTE; the σCDFA is shown in Figure 6 (Middle). Finally, the difference σCDFA is generated as shown in Figure 6 (Bottom).

We see from Figure 6 (Bottom) that even though the two other σCDFAs contain indeterminate transitions, we can verify language equivalence, because the difference language is empty—Figure 6 (Bottom) contains no accepting states.

We were able to verify that our testing flow works, and were unable to find bugs in our RTE and σCDFA functions. As an anecdote, we did find a bug in our random σCDFA generator, `gen-dfa`, which was sometimes labeling transitions in a way which violated determinism.

The reader might question how good our σCDFA generator is. How uniform is the selection of language? It is unclear what the best way is to measure this uniformity. Nevertheless, we provide some basic statistics.

First, we asked whether the language of a generated σCDFA was inhabited. The `gen-dfa` function generated 1144 σCDFAs (23.19%) with inhabited languages and 3767 (76.35%) indeterminate.

In the second set of simulations we generated pairs (A, B) of σCDFAs to measure the likelihood of languages being subsets or disjoint. We found $\mathbb{P}[\mathcal{L}\{A\} \subset \mathcal{L}\{B\}] = 14.18\%$, $\mathbb{P}[\mathcal{L}\{A\} \not\subset \mathcal{L}\{B\}] = 23.87\%$, and $\mathbb{P}[\texttt{dont-know}] = 61.95\%$. Additionally, we found $\mathbb{P}[\mathcal{L}\{A\} \cap \mathcal{L}\{B\} = \emptyset] = 50.99\%$, $\mathbb{P}[\mathcal{L}\{A\} \cap \mathcal{L}\{B\} \neq \emptyset] = 5.81\%$, and $\mathbb{P}[\texttt{dont-know}] = 43.21\%$.

Figure 7 shows various measurements as a function of the number of states in the σCDFA. A few takeaways are that (1) roughly 15 to 20% of the σCDFAs generated are provably inhabited, (2) with roughly for 80% the habitation is indeterminate. (3) Between 40 and 50% of the time it cannot be determined whether the languages of two σCDFAs intersect. (4) Roughly 15 to 25% of the time it can be proven that one language is a subset of another, and (5) 0 to 5% of the time that two languages have an inhabited intersection.

It may seem counter-intuitive that the curve labeled `overlap=true` should lie below `subset=true`. But recall $A \subset B \implies (A \cap B \neq \emptyset)$. In particular, $A \subset B$, whenever $A = \emptyset$. Figure 8 shows another enlightening example. In the figure we see σCDFAs A and B. To compute whether A and B overlap, we compute their intersection, $A \cap B$. $\mathcal{L}\{A \cap B\}$ has indeterminate habitation, because the only path from $q_{0\times0}$ to an accepting state $q_{2\times2}$ has a transition labeled t_3 which is indeterminate. However, we know that $\mathcal{L}\{A\} \subset \mathcal{L}\{B\}$, because the automaton $A \cap \overline{B}$ has no final state; thus $\mathcal{L}\{A \cap \overline{B}\} = \emptyset$.

The raw data and code for generation/analysis are publicly avaialble here:

`github.com/jimka2001/clojure-rte/tree/master/resources/statistics`
`github.com/jimka2001/clojure-rte/tree/master/src/statistics.clj`

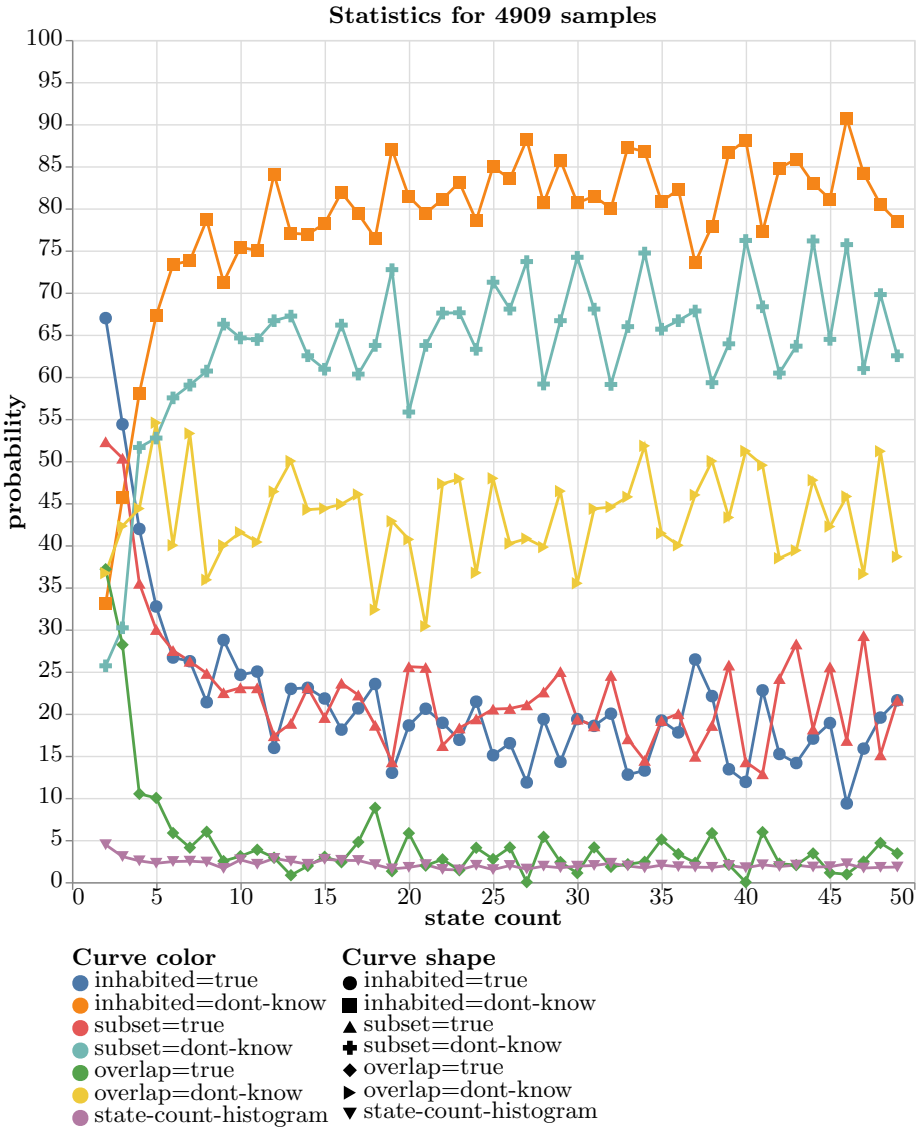
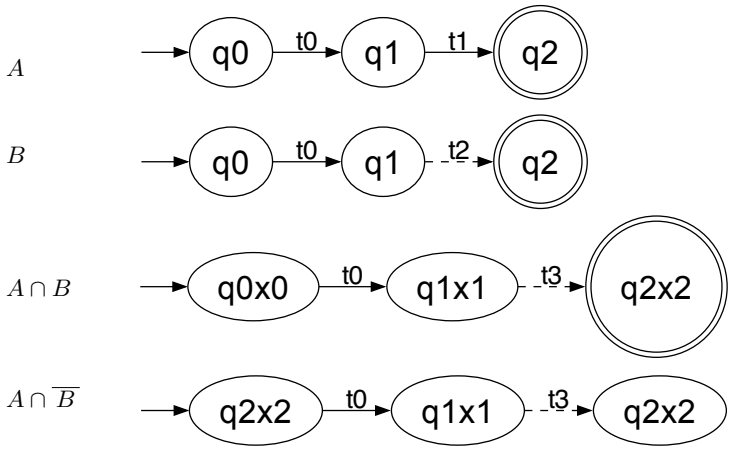


Fig. 7: Generation summary of 4909 σCDFAs: probabilities of habitation, overlap, and subsetness as a function of state count. The purple, mostly flat curve (State–count-histogram), running between 0 and 3% shows a histogram, indicating the percentage of total samples (y-axis) for a given (x-axis) state count. That the curve is relatively flat shows a relatively evenly distributed sampling of σCDFAs as measured 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))
```

Fig. 8: σCDFAs A and B, together with $A \cap B$ and $A \cap \overline{B}$. We see the habitation of $\mathcal{L}\{A \cap B\}$ is indeterminate (the only successful computation contains t_3 which is indeterminate), while the $\mathcal{L}\{A \cap \overline{B}\}$ is empty (there are no final/accepting states). Values of $t_0 \ldots t_3$ are displayed in Clojure s-expression [6] syntax.

5 Conclusion and Perspectives

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.

Future work will include expanding our testing framework to be more theoretically sound, and contrasting with the current engineering approach.

Constructing a tree structure randomly (as in Section 4.2) 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 [19] 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.

ChatGPT contributed no scientific content to this paper; however, it did provide invaluable proofreading and editorial support, making the text clearer and more compelling than our original draft. We hereby make the relevant chat open and public: https://chatgpt.com/share/687e3798-22e0-800d-9d34-e4a27a2ecd32.

References

1. Seqexp: regular expressions for sequences (2014), https://github.com/cgrand/seqexp
2. Ansi: American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999) (1994)
3. Broda, S., Holzer, M., Maia, E., Moreira, N., Reis, R.: A mesh of automata. Information and Computation **265**, 94–111 (2019). https://doi.org/https://doi.org/10.1016/j.ic.2019.01.003, https://www.sciencedirect.com/science/article/pii/S0890540119300033

4. Brzozowski, J.A.: Derivatives of Regular Expressions. J. ACM **11**(4), 481–494 (Oct 1964). https://doi.org/10.1145/321239.321249, http://doi.acm.org/10.1145/321239.321249
5. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. p. 268–279. ICFP '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/351240.351266, https://doi.org/10.1145/351240.351266
6. Dai, H.Y.: The mysteries of lisp – i: The way to s-expression lisp (2015), https://arxiv.org/abs/1505.07375
7. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Computer Aided Verification, 29th International Conference (CAV'17). Springer (July 2017), https://www.microsoft.com/en-us/research/publication/power-symbolic-automata-transducers-invited-tutorial/
8. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische mathematik **1**(1), 269–271 (1959)
9. Fink, G., Bishop, M.: Property-based testing: A new approach to testing for assurance. SIGSOFT Softw. Eng. Notes **22**(4), 74–80 (Jul 1997). https://doi.org/10.1145/263244.263267, https://doi.org/10.1145/263244.263267
10. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM **55**(4), 19:1–19:64 (Sep 2008). https://doi.org/10.1145/1391289.1391293, http://doi.acm.org/10.1145/1391289.1391293
11. Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 8 Edition. Addison-Wesley Professional, 1st edn. (2014)
12. Grigore, R.: Java generics are turing complete. CoRR **abs/1605.05274** (2016), http://arxiv.org/abs/1605.05274
13. Heikkilä, M.: Malli, Metosin (2022), https://github.com/metosin/malli
14. Hickey, R.: The Clojure Programming Language. In: Proceedings of the 2008 symposium on Dynamic languages. p. 1. ACM (2008)
15. Hickey, R.: A History of Clojure. Proc. ACM Program. Lang. **4**(HOPL) (Jun 2020). https://doi.org/10.1145/3386321, https://doi.org/10.1145/3386321
16. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
17. Keil, M., Thiemann, P.: Symbolic Solving of Extended Regular Expression Inequalities. In: Raman, V., Suresh, S.P. (eds.) 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014). Leibniz International Proceedings in Informatics (LIPIcs), vol. 29, pp. 175–186. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2014). https://doi.org/10.4230/LIPIcs.FSTTCS.2014.175, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSTTCS.2014.175
18. Kennedy, A., Pierce, B.C.: On decidability of nominal subtyping with variance. In: International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD) (January 2007), https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/
19. Koechlin, F., Rotondo, P.: Absorbing Patterns in BST-Like Expression-Trees. In: Bläser, M., Monmege, B. (eds.) 38th International Symposium on Theoretical Aspects of Computer Science (STACS 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 187,

pp. 48:1–48:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.STACS.2021.48, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2021.48
20. Might, M., Darais, D., Spiewak, D.: Parsing with derivatives: A functional pearl. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. p. 189–195. ICFP '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2034773.2034801, https://doi.org/10.1145/2034773.2034801
21. Miller, A.: Spec Guide (2022), https://clojure.org/guides/spec
22. Newton, J.: Representing and Computing with Types in Dynamically Typed Languages. Ph.D. thesis, Sorbonne University (Nov 2018)
23. Newton, J.: An elegant and fast algorithm for paritioning types. In: European Lisp Symposium. Amsterdam, Netherlands (Apr 2023)
24. Newton, J.: Regular Type Expressions for Clojure (2024), github.com/jimka2001/clojure-rte
25. Newton, J.: Regular Type Expressions for Common Lisp (2024), github.com/jimka2001/cl-rte
26. Newton, J.: Regular Type Expressions for Python (2024), github.com/jimka2001/python-rte
27. Newton, J.: Regular Type Expressions for Scala (2024), github.com/jimka2001/scala-rte
28. Newton, J.: Type-checking heterogeneous sequences in a simple embeddable type system. In: Erdem, E., Vidal, G. (eds.) Practical Aspects of Declarative Languages. pp. 18–34. Springer Nature Switzerland, Cham (2025)
29. Newton, J., Pommellet, A.: A portable, simple, embeddable type system. In: European Lisp Symposium. Online, Everywhere (May 2021)
30. Newton, J., Verna, D.: Recognizing hetergeneous sequences by rational type expression. In: Proceedings of the Meta'18: Workshop on Meta-Programming Techniques and Reflection. Boston, MA USA (Nov 2018)
31. Newton, J., Verna, D.: Finite automata theory based optimization of conditional variable binding. In: European Lisp Symposium. Genova, Italy (Apr 2019)
32. Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: The Scala language specification (2004)
33. Odersky, M., Zenger, M.: Scalable component abstractions. In: Sigplan Notices - SIGPLAN. vol. 40, pp. 41–57 (10 2005). https://doi.org/10.1145/1103845.1094815
34. Owens, S., Reppy, J., Turon, A.: Regular-expression Derivatives Re-examined. J. Funct. Program. **19**(2), 173–190 (Mar 2009)
35. van Rossum, G., Drake, F.L.: The Python Language Reference Manual. Network Theory Ltd. (2011)
36. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press, USA (2009)