For submission to the Journal of Automata, Languages and Combinatorics
Created on December 16, 2017

SYMBOLIC MANIPULATION OF CODE PROPERTIES

Stavros Konstantinidis[A,C] Casey Meijer[A] Nelma Moreira[B,D]
Rogério Reis[B,D]

[A] *Department of Mathematics and Computing Science*
Saint Mary's University, 923 Robie Str.
Halifax, Nova Scotia, B3H 3C3, Canada
s.konstantinidis@smu.ca (S. Konstantinidis)
dylanyoungmeijer@gmail.com (C. Meijer)

[B] *CMUP & DCC, Faculdade de Ciências da Universidade do Porto*
Rua do Campo Alegre, 4169-007 Porto, Portugal
nam@dcc.fc.up.pt (N. Moreira)
rvr@dcc.fc.up.pt (R. Reis)

ABSTRACT

The FAdo system is a symbolic manipulator of formal languages objects, implemented in Python. In this work, we extend its capabilities by implementing methods to manipulate transducers and we go one level higher than existing formal language systems and implement methods to manipulate objects representing classes of independent languages (widely known as code properties). Our methods allow users to define their own code properties and combine them between themselves or with fixed properties such as prefix codes, suffix codes, error detecting codes, etc. The satisfaction and maximality decision questions are solvable for any of the definable properties. The new online system LaSer allows to query about code properties and obtain the answer in a batch mode. Our work is founded on independence theory as well as the theory of rational relations and transducers, and contributes with improved algorithms on these objects.

Keywords: algorithms, automata, codes, FAdo, implementation, independence, LaSer, maximal, regular languages, transducers, program generation

1. Introduction

Several programming platforms exist providing methods to transform and manipulate various formal language objects: Grail/Grail+ [28,35], Vaucanson [6,36], FAdo [2,10], OpenFST [20], JFLAP [20]. Some of these systems allow one to manipulate such objects within simple script environments. Grail for example, one of the oldest systems, provides a set of filters manipulating automata and regular expressions on a UNIX

[C] Research supported by NSERC.
[D] Research supported by CMUP (UID/MAT/00144/2013), FCT through FEDER and PT2020.

command shell. Similarly, FAdo provides a set of methods manipulating such objects on a Python shell [27]. Software environments for symbolic manipulation of formal languages are widely recognized as important tools for theoretical and applied research. They allow easy prototyping of new algorithms, testing algorithm performance with large datasets, corroborate or disprove descriptional complexity bounds for manipulations of formal systems representations, etc. A typical example is, for a given operation on regular languages, to find an upper bound for the number of states of a minimal deterministic finite automaton (DFA) for the language that results from the operation, as a function of the number of states of the minimal DFAs of the operands. Due to the combinatorial nature of formal languages representations, this kind of calculations are almost impossible without computational aid.

In this work, we extend the capabilities of FAdo and LaSer [9,18] by implementing transducer methods and by going to the higher level of implementing objects representing classes of independent formal languages, also known as code properties. More specifically, the contributions of the present paper are as follows.

> (I) Implementation of transducer objects and associated methods: rational transducer operations, product constructions between transducers and between transducers and automata, as well as a transducer functionality test. (II) Definitions of objects representing code properties and methods for their manipulation, which to our knowledge is a new development in software related to formal language objects. (III) Enhancement and implementation of decision algorithms for code properties of regular languages. In particular, many such algorithms have been implemented and enhanced so as to provide witnesses (counterexamples) in case of a negative answer, for example, when the given regular language does not satisfy the property, or is not maximal with respect to the property. To our knowledge such implementations are not openly available. (IV) A mathematical definition of what it means to simulate (and hence implement) a hierarchy of properties and the proof that there is no complete simulation of the set of error-detecting properties. (V) Generation of executable Python code based on the user's requested question about a given code property.

Our work is founded on dependence theory [15, 34] as well as the theory of rational relations and transducers [4, 30].

The paper is organized as follows. Section 2 contains basic terminology about various formal language concepts as well as a few examples of manipulating FAdo automata. Section 3 describes our implementation of transducer objects, rational transducer operations, and product constructions. Section 4 describes an existing decision algorithm for transducer functionality, and then our enhancement so as to provide witnesses when the transducer in question is not functional. Section 5 describes our implementation of code property objects and their manipulation, as well as a mathematical approach to defining simulations of infinite sets of properties and proving that a simulation of all error-detecting properties exists, but no complete such simulation is possible. Section 6 continues on code property methods by describing our implementation of the satisfaction and maximality methods. Again, we describe our enhancements so as to provide witness version of these questions. Section 7 concerns

the unique decodability (or decipherability) property and its satisfaction and maximality algorithms. Section 8 discusses briefly the new version of LaSer. Section 9 contains a few concluding remarks including directions for future research.

2. Terminology and Background

Sets, alphabets, words, languages. We write $\mathbb{N}, \mathbb{N}_0$ for the sets of natural numbers (not including 0) and non-negative integers, respectively. If S is a set, then $|S|$ denotes the cardinality of S, and 2^S denotes the set of all subsets of S. An *alphabet* is a finite nonempty set of symbols. In this paper, we write Σ, Δ for any arbitrary alphabets. The set of all words, or strings, over an alphabet Σ is written as Σ^*, which includes the *empty word* ε. A *language* (over Σ) is any set of words. Let L be a language and let u, v, w, x be any words. If $w \in L$ then we say that w is an *L-word*. When there is no risk of confusion, we write a singleton language $\{w\}$ simply as w. For example, $L \cup w$ and $v \cup w$ mean $L \cup \{w\}$ and $\{v\} \cup \{w\}$, respectively. We use standard operations and notation on words and languages [13, 22, 29, 38]. If w is of the form uv then u is a *prefix* and v is a *suffix* of w. If w is of the form uxv then x is an *infix* of w. If $u \neq w$ then u is called a *proper prefix* of w—the definitions of proper suffix and proper infix are similar.

Codes, properties, independent languages, maximality. A *property* (over Σ) is any set $\mathcal{P}$ of languages. If L is in $\mathcal{P}$ then we say that L *satisfies* $\mathcal{P}$. A *code property*, or *independence*, [15], is a property $\mathcal{P}$ for which there is $n \in \mathbb{N} \cup \{\aleph_0\}$ such that

$$L \in \mathcal{P}, \quad \text{if and only if} \quad L' \in \mathcal{P}, \text{ for all } L' \subseteq L \text{ with } 0 < |L'| < n,$$

that is, L satisfies $\mathcal{P}$ exactly when all nonempty subsets of L with less than n elements satisfy $\mathcal{P}$. In this case, we also say that $\mathcal{P}$ is an *n-independence*. In the rest of the paper we only consider properties $\mathcal{P}$ that are code properties. A language $L \in \mathcal{P}$ is called *$\mathcal{P}$-maximal*, or a maximal $\mathcal{P}$ code, if $L \cup w \notin \mathcal{P}$ for any word $w \notin L$. Every language satisfying $\mathcal{P}$ is included in some $\mathcal{P}$-maximal language [15]. To our knowledge, all known code related properties in the literature [5, 7, 9, 11, 15, 25, 33, 40] are code properties as defined here. For example, consider the 'prefix code' property: L is a *prefix code* if no word in L is a proper prefix of a word in L. This is a code property with $n = 3$. As we shall see further below the focus of this work is on 3-independence properties that can also be viewed as independent with respect to a binary relation in the sense of [34].

Automata and regular languages [30, 39]. A nondeterministic finite automaton with empty transitions, for short *automaton* or *ε-NFA*, is a quintuple $\mathbf{a} = (Q, \Sigma, T, I, F)$ such that Q is the set of states, Σ is an alphabet, $I, F \subseteq Q$ are the sets of start (or initial) states and final states, respectively, and $T \subseteq Q \times (\Sigma \cup \varepsilon) \times Q$ is the finite set of *transitions*. If (p, x, q) is a transition, then x is the *label* of the transition, and we say that p has an *outgoing* transition (with label x). A *path* of $\mathbf{a}$ is a finite sequence $(p_0, x_1, p_1, \ldots, x_\ell, p_\ell)$, for some $\ell \in \mathbb{N}_0$, such that each (p_{i-1}, x_i, p_i) is a transition of $\mathbf{a}$. The word $x_1 \cdots x_\ell$ is called the *label* of the path. The path is called

accepting if p_0 is a start state and p_ℓ is a final state. The *language accepted* by $\mathbf{a}$, denoted as $\mathrm{L}(\mathbf{a})$, is the set of labels of all the accepting paths of $\mathbf{a}$. The automaton $\mathbf{a}$ is called *trim*, if every state appears in some accepting path of $\mathbf{a}$. It is called an *NFA*, if no transition label is empty, that is, $T \subseteq Q \times \Sigma \times Q$. It is called a deterministic finite automaton, or *DFA* for short, if I is a singleton set and there is no state p having two outgoing transitions with equal labels. The *size* $|\mathbf{a}|$ of the automaton $\mathbf{a}$ is $|Q| + |T|$. The automaton $\mathbf{a}^\varepsilon$ results when we add *ε-loops* in $\mathbf{a}$, that is, transitions (p, ε, p) for all states $p \in Q$. Then $\mathrm{L}(\mathbf{a}) = \mathrm{L}(\mathbf{a}^\varepsilon)$.

Transducers and (word) relations [4, 30, 39]. A (word) *relation* over Σ and Δ is a subset of $\Sigma^* \times \Delta^*$, that is, a set of pairs (x, y) of words over the two alphabets (respectively). The *inverse* of a relation ρ, denoted as ρ^{-1}, is the relation $\{(y, x) \mid (x, y) \in \rho\}$. A *transducer* is a sextuple $\mathbf{t} = (Q, \Sigma, \Delta, T, I, F)$ such that Q, I, F are exactly the same as those in ε-NFAs, Σ is now called the *input* alphabet, Δ is the *output* alphabet, and $T \subseteq Q \times \Sigma^* \times \Delta^* \times Q$ is the finite set of transitions. We write $(p, x/y, q)$ for a transition—the *label* (x/y) consists of the input label x and the output label y. The concepts of path, accepting path, and trim transducer are similar to those in ε-NFAs. In particular, the *label* of a path $(p_0, x_1/y_1, p_1, \ldots, x_\ell/y_\ell, p_\ell)$ is the pair $(x_1 \cdots x_\ell, y_1 \cdots y_\ell)$ made by concatenating the input labels, and the output labels occurring in the path. The *relation realized* by the transducer $\mathbf{t}$, denoted as $\mathrm{R}(\mathbf{t})$, is the set of labels of all the accepting paths of $\mathbf{t}$. We write $\mathbf{t}(x)$ for the set of *possible outputs of* $\mathbf{t}$ on input x, that is, $y \in \mathbf{t}(x)$ iff $(x, y) \in \mathrm{R}(\mathbf{t})$. For any language L, $\mathbf{t}(L)$ is the language $\cup_{x \in L} \mathbf{t}(x)$. The *domain* of $\mathbf{t}$ is the set of all words w such that $\mathbf{t}(w) \neq \emptyset$. The *inverse* of $\mathbf{t}$, denoted as $\mathbf{t}^{-1}$, is the transducer that results from $\mathbf{t}$ by simply switching the input with the output alphabet of $\mathbf{t}$ and also switching the input with the output label in each transition of $\mathbf{t}$. Then, $\mathbf{t}^{-1}$ realizes the relation $\big(\mathrm{R}(\mathbf{t})\big)^{-1}$. The transducer $\mathbf{t}$ is said to be in *standard form*, if each transition $(p, x/y, q)$ is such that $x \in (\Sigma \cup \varepsilon)$ and $y \in (\Delta \cup \varepsilon)$. It is in *normal form* if it is in standard form and exactly one of x and y is equal to ε. We note that every transducer is effectively equivalent to one (realizing the same relation, that is) in standard form and one in normal form. As in the case of automata, the transducer $\mathbf{t}^\varepsilon$ results when we add *ε-loops* in $\mathbf{t}$, that is, transitions $(p, \varepsilon/\varepsilon, p)$ for all states $p \in Q$. Then, $\mathrm{R}(\mathbf{t}) = \mathrm{R}(\mathbf{t}^\varepsilon)$. The *size* of a transition $(p, x/y, q)$ is the number $1 + |x| + |y|$. The size $|\mathbf{t}|$ of the transducer $\mathbf{t}$ is the sum of the number of states and sizes of transitions in T. If $\mathbf{s}$ and $\mathbf{t}$ are transducers, then there is a transducer $\mathbf{s} \vee \mathbf{t}$ of size $O(|\mathbf{s}| + |\mathbf{t}|)$ realizing $\mathrm{R}(\mathbf{s}) \cup \mathrm{R}(\mathbf{t})$.

Automata and finite languages in FAdo [10]. FAdo contains the modules `fa` for automata, `fl` for finite languages, and `fio` for input/output of formal language objects, which can be imported in a standard Python manner. The FAdo object classes `FL`, `DFA` and `NFA` manipulate finite languages, DFAs and ε-NFAs, respectively.

Example 1. The following code uses the class name `FL` of the module `fl` to define the finite language `L` from the given list of strings, and then defines an NFA object `a` accepting the language `L`, which is $\{a, ab, aab\}$.

```
import FAdo.fl as fl
```

```
import FAdo.fio as fio
from FAdo.fa import *    # import all fa methods for readability
lst = ['a', 'ab', 'aab']
L = fl.FL(lst)
a = L.toNFA()
st = '@NFA 1 * 0\n0 a 0\n0 b 1\n'
b = fio.readOneFromString(st)
```

The second last line defines a string `st` containing the description of an automaton in FAdo format, [10,18], which accepts a^*b, and then uses `st` to define the `NFA` object `b`. The string `st` contains three lines: the first indicates the type of object followed by the final states (in this case `1`) and the start states after `*` (in this case `0`); the second line contains the transition `0 a 0`; and the third line contains the transition `0 b 1`. One can also use `fio.readOneFromFile(file)` to read an automaton, or transducer, from `file`. □

Example 2. Assume that `a` is a FAdo automaton and `w` is a string (a word). The method `a.evalWordP(w)` returns whether `a` accepts `w`. The following code shows a quick implementation of `a.evalWordP(w)`

```
b = fl.FL([w]).toNFA()
c = a & b
return not c.emptyP()
```

One verifies that `w` is accepted by `a` if and only if `a` intersected with any automaton accepting {`w`} accepts something. Here, `a & b` returns an NFA accepting L(`a`) ∩ L(`b`). Both `a` and `b` must be NFAs with no ε-transitions. Method `c.emptyP()` returns whether the language accepted by `c` is empty. □

3. Transducer Object Classes and Methods

In this section we discuss some aspects of the implementation of transducer objects and some of their methods: product constructions and rational operations. We discuss the method for testing functionality in Section 4. The module containing all that is discussed in this and the next section is called `transducers.py`.

Transducer objects and basic methods We implement the class `GFT`, for General Form Transducer, as a subclass of `NFA`. A transducer $\mathbf{t} = (Q, \Sigma, \Delta, T, I, F)$ is implemented as an object `t` with six instance variables `States, Sigma, Output, delta, Initial, Final` corresponding to the six components of $\mathbf{t}$. Standard form transducers are objects of the FAdo class `SFT`, which is a subclass of `GFT`. The class `SFT` is very important from an algorithmic point of view, as most product constructions require a transducer to be in standard form. The conversion from GFT to SFT is done using the method `t.toSFT()` which returns an SFT transducer equivalent to `t`. The implementation of Normal Form Transducers is via the FAdo class `NFT`. This form of transducers is convenient in proving mathematical statements about transducers [30].

Example 3. The following code defines a string `s` containing a transducer description, and then constructs an SFT transducer from the string `s`. The transducer, on input x, returns any proper suffix of x—see also Fig 1.

```
s = '@Transducer 1 * 0\n'\
    '0 a @epsilon 0\n0 b @epsilon 0\n'\
    '0 a @epsilon 1\n0 b @epsilon 1\n'\
    '1 a a 1\n1 b b 1\n'
t = fio.readOneFromString(s)
a = t.runOnWord('ababb')
n = len('ababb')
print a.enumNFA(n)
```

Method `t.runOnWord(w)` assumes that `t` is an SFT object and returns an automaton accepting the language `t(w)`—recall, this is the set of possible outputs of `t` on input `w`. The last statement prints the set of all proper suffixes of the word `ababb`. □

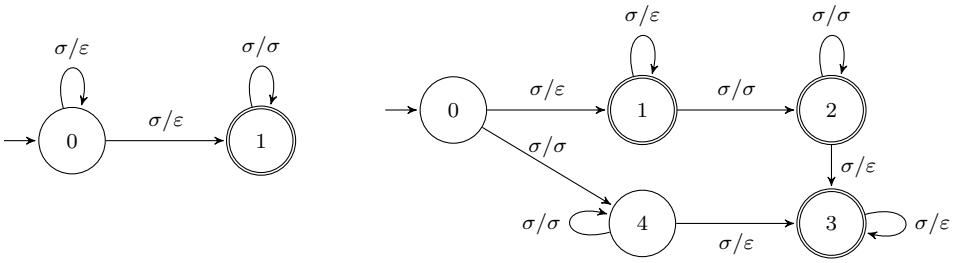


Figure 1: On input x, the left transducer outputs any proper suffix of x. The right transducer outputs any proper infix of x. **Note**: In this and the following transducer figures, the input and output alphabets are equal. An arrow with label σ/σ represents a set of transitions with labels σ/σ, for all alphabet symbols σ; and similarly for an arrow with label σ/ε. An arrow with label σ/σ' represents a set of transitions with labels σ/σ' for all distinct alphabet symbols σ, σ'.

Assuming again that `t` is an SFT object, we have the following methods.

`t.inverse()`: returns the inverse of the transducer `t`.

`t.evalWordP((u,v))`: returns whether the pair `(u,v)` belongs to the relation realized by `t`, or equivalently whether `v` $\in$ `t(u)`. Other useful methods are as follows. `t.nonEmptyW()`: returns some word pair `(u, v)` which belongs to the relation realized by `t`, if nonempty, else it returns the pair `(None, None)`. `t.toInNFA()`: returns the NFA that results if we remove the output alphabet and the output labels of the transitions in `t`. `t.toOutNFA()`: returns the NFA that results if we remove the input alphabet and the input labels of the transitions in `t`.

Product constructions [4, 16, 39] The following methods are available in FAdo. They are adaptations of the standard product construction [13] between two NFAs

which produces an NFA with transitions $((p_1, p_2), \sigma, (q_1, q_2))$, where (p_1, σ, q_1) and (p_2, σ, q_2) are transitions of the two NFAs, such that the new NFA accepts the intersection of the corresponding languages. We assume that `t` and `s` are SFT objects and `a` is an NFA object.

`t.inIntersection(a)`: returns a transducer realizing all word pairs (x, y) such that x is accepted by `a` and (x, y) is realized by `t`.

`t.outIntersection(a)`: returns a transducer realizing all word pairs (x, y) such that y is accepted by `a` and (x, y) is realized by `t`.

`t.runOnNFA(a)`: returns an automaton that accepts $\mathbf{t}\big(\mathrm{L}(\mathbf{a})\big) = \bigcup_{x \in \mathrm{L}(\mathbf{a})} \mathbf{t}(x)$.

`t.composition(s)`: returns a transducer realizing the composition $\mathrm{R}(\mathtt{t}) \circ \mathrm{R}(\mathtt{s})$ of the relations realized by the transducers `t` and `s`.

Rational operations [4] A relation ρ is a *rational relation*, if it is equal to $\emptyset$, or to $\{(x, y)\}$ for some words x and y, or it can be obtained from other ones by using a finite number of times any of the three (rational) operators: union, concatenation, Kleene star. We have implemented those rational operators as `t.union(s)`, `t.concat(s)`, `t.star()`. A classic result on transducers says that a relation is rational if and only if it can be realized by a transducer.

4. Witness of Transducer *non*-functionality

A transducer $\mathbf{t}$ is called *functional* if, for every word w, the set $\mathbf{t}(w)$ is either empty or a singleton. A triple of words (w, z, z') is called a *witness of* $\mathbf{t}$*'s non-functionality*, if $z \neq z'$ and $z, z' \in \mathbf{t}(w)$. In this section we present the SFT method `t.nonFunctionalW()`, which returns a witness of `t`'s non-functionality, or the triple (`None,None,None`) if `t` is functional. The method is an adaptation of the decision algorithms in [1, 3] that return whether a given transducer in standard form is functional. Although there are some differences in the two algorithms, we believe that conceptually the algorithmic technique is the same. We first describe in two phases that algorithmic technique following the presentation in [3], and then we modify it in order to produce the method `t.nonFunctionalW()`. We also note that, using a careful implementation and assuming fixed alphabets, the time complexity of the decision algorithm can be quadratic with respect to the size of the transducer [1].

Given a standard form transducer $\mathbf{t} = (Q, \Sigma, \Delta, T, I, F)$, the first phase is to construct the *square machine* $\mathbf{u}$, which is defined by the following process.

Phase 1

(I) First define an automaton $\mathbf{u}'$ as follows: states $Q \times Q$, initial states $I \times I$, and final states $F \times F$.

(II) If $\mathbf{t}$ contains ε-input transitions, that is, transitions with labels of the form ε/u then we let $\mathbf{t}$ be $\mathbf{t}^{\varepsilon}$. The transitions of $\mathbf{u}'$ are all the triples

$$((p, p'), (x, x'), (q, q'))$$

such that $(p, v/x, q)$ and $(p', v/x', q')$ are transitions of $\mathbf{t}$.

(iii) Return $\mathbf{u}$ = a trim version of $\mathbf{u}'$.

Note that any accepting path of $\mathbf{u}$ has a label $(x_1, x_1') \cdots (x_n, x_n')$ such that the words $x_1 \cdots x_n$ and $x_1' \cdots x_n'$ are outputs (possibly equal) of $\mathbf{t}$ on the *same* input word. The next phase is to perform a process that starts from the initial states and assigns a *delay value* to each state, which is either ZERO or a pair of words in $\{(\varepsilon, \varepsilon), (\varepsilon, u), (u, \varepsilon)\}$, with u being nonempty. A delay (y, y') on a state (p, p') indicates that there is a path in $\mathbf{u}$ from $I \times I$ to (p, p') whose label is a word pair of the form (fy, fy'). This means that there is an input word that can take the transducer $\mathbf{t}$ to state p with output fy and also to state p' with output fy'. A delay ZERO at (p, p') means that there is an input word that can take $\mathbf{t}$ to state p with output of the form $f\sigma g$ and to state p' with output of the form $f\sigma' g'$, where σ and σ' are distinct alphabet symbols.

Phase 2

(iv) Assign to each initial state the delay value $(\varepsilon, \varepsilon)$.

(v) Starting from the initial states, visit all transitions in breadth-first search mode such that, if (p, p') has delay value (y, y') and a transition $((p, p'), (x, x'), (q, q'))$ is visited, then the state (q, q') gets a delay value D as follows:

- If $y'x'$ is of the form yxu then $D = (\varepsilon, u)$. If yx is of the form $y'x'u$ then $D = (u, \varepsilon)$. If $y'x' = yx$ then $D = (\varepsilon, \varepsilon)$. Else, D = ZERO.

(vi) The above process stops when a delay value is ZERO, or a state gets two different delay values, or every state gets one delay value.

(vii) If every state has one delay value and every final state has the delay value $(\varepsilon, \varepsilon)$ then return `True` (the transducer is functional). Else, return `False`.

Next we present our witness version of the transducer functionality algorithm. First, the square machine $\mathbf{u}$ is *revised* such that its transitions are of the form

$$((p, p'), (v, x, x'), (q, q')),$$

that is, we now record in $\mathbf{u}$ information about the common input v (see Step 2 in Phase 1). Then, to each state (q, q') we assign not only a delay value but also a path value (α, β, β') which means that, on input α, the transducer $\mathbf{t}$ can reach state q with output β and also state q' with output β'—see Fig. 2.

Definition 4. Let (q, q') be a state of the revised square machine $\mathbf{u}$. The set of *delay-path values* of (q, q') is defined as follows.

- If (q, q') is an initial state then $((\varepsilon, \varepsilon), (\varepsilon, \varepsilon, \varepsilon))$ is a delay-path value of (q, q').
- If $((p, p'), (v, x, x'), (q, q'))$ is a transition in $\mathbf{u}$ and (p, p') has a delay-path value $(C, (w, z, z'))$, then $(D, (wv, zx, z'x'))$ is a delay-path value of (q, q'), where D is defined as follows.
  - (i) If $C = (y, y') \neq$ ZERO and $y'x'$ is of the form yxu then $D = (\varepsilon, u)$.
  - (ii) If $C = (y, y') \neq$ ZERO and yx is of the form $y'x'u$ then $D = (u, \varepsilon)$.
  - (iii) If $C = (y, y') \neq$ ZERO and $y'x' = yx$ then $D = (\varepsilon, \varepsilon)$
  - (iv) Else, D = ZERO.

For (q, q'), we also define a *suffix triple* $(w_{qq'}, z_{qq'}, z'_{qq'})$ to be the label of any path from (q, q') to a final state of $\mathbf{u}$.

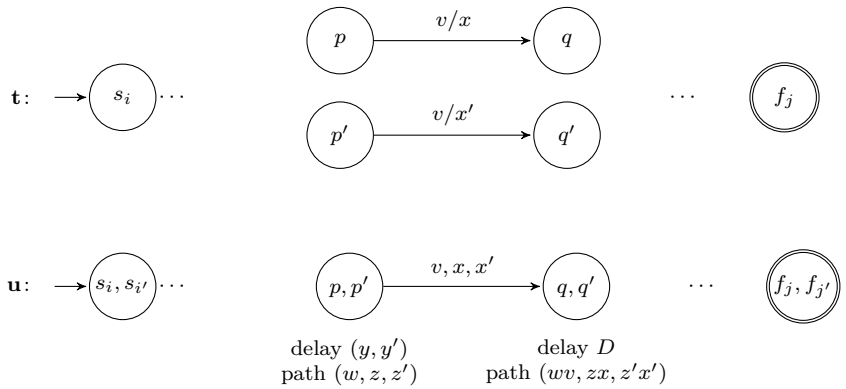


Figure 2: The transducer **t** and the corresponding (revised) square machine **u**.

Remark 5. The above definition implies that if a state (p, p') has a delay-path value $(C, (w, z, z'))$, then there is a path in **u** whose label is (w, z, z'). Moreover, by the definition of **u**, the transducer **t** on input w can reach state p with output z and also state p' with output z'. Thus, if (p, p') is a final state, then $z, z' \in \mathbf{t}(w)$.

Algorithm nonFunctionalW

(I) Define a function `completePath`(q, q') that follows a shortest path from (q, q') to a final state of **u** and returns a suffix triple (see Definition 4).

(II) Construct the revised square machine **u**, as in Phase 1 above but now use transitions of the form $((p, p'), (v, x, x'), (q, q'))$—see step 2 in Phase 1.

(III) Assign to each initial state the delay-path value $((\varepsilon, \varepsilon), (\varepsilon, \varepsilon, \varepsilon))$.

(IV) Starting from the initial states, visit all transitions in breadth-first search mode. If (p, p') has delay-path value $((y, y'), (w, z, z'))$, and a transition $((p, p'), (v, x, x'), (q, q'))$ is visited, then compute the delay value D of (q, q') as in steps 1–4 of Definition 4, and let $R = (wv, zx, z'x')$. Then,

- (A) if D is ZERO, then invoke `completePath`(q, q') to get a suffix triple $(w_{qq'}, z_{qq'}, z'_{qq'})$ and `return` $(wvw_{qq'}, zxz_{qq'}, z'x'z'_{qq'})$.
- (B) if (q, q') is final and $D \neq (\varepsilon, \varepsilon)$, `return` $(wv, zx, z'x')$.
- (C) if (q, q') already has a delay value $\neq D$ and, hence, a path value $P = (w_1, z_1, z'_1)$, then invoke `completePath`(q, q') to get a suffix triple $(w_{qq'}, z_{qq'}, z'_{qq'})$. Then,
  - If $zxz_{qq'} \neq z'x'z'_{qq'}$ `return` $(wvw_{qq'}, zxz_{qq'}, z'x'z'_{qq'})$.
  - Else `return` $(w_1w_{qq'}, z_1z_{qq'}, z'_1z'_{qq'})$.
- (D) else assign (D, R) to (q, q') as delay-path value and continue the breadth-first process.

(V) Return `(None,None,None)`, as the breadth-first process has been completed.

Terminology. Let $A = (w_1, \ldots, w_k)$ be a tuple consisting of words. The *size* $|A|$ of A is the number $\sum_{i=1}^{k} |w_i| + (k-1)$. For example, $|(0, 01, 10)| = 7$. If $\{A_i\}$ is any set of word tuples then a *minimal* element (of that set) is any A_i whose size is the minimum of $\{|A_i|\}$.

Theorem 6. *If algorithm* `nonFunctionalW` *is given as input a standard form transducer* $\mathbf{t}$*, then it returns either a size* $O(|\mathbf{t}|^2)$ *witness of* $\mathbf{t}$*'s non-functionality, or the triple* (`None,None,None`) *if* $\mathbf{t}$ *is functional.*

Before we proceed with the proof of the above result, we note that there is a sequence $(\mathbf{t}_i)$ of non-functional transducers such that $|\mathbf{t}_i| \to \infty$ and any minimal witness of $\mathbf{t}_i$'s non-functionality is of size $\Theta(|\mathbf{t}_i|^2)$. Indeed, let (p_i) be the sequence of primes in increasing order and consider the transducer $\mathbf{t}_i$ shown in Fig. 3. It has size $\Theta(p_i)$ and every output word w of $\mathbf{t}_i$ has length equal to that of the input used to get w. The relation realized by $\mathbf{t}_i$ is

$$\{(0^{mp_i}, 0^{mp_i}),\ (0^{n(p_i+1)}, 10^{n(p_i+1)-1}) \ :\ m, n \in \mathbb{N}\}.$$

Any minimal witness of $\mathbf{t}_i$'s non-functionality is of the form $w_i = (0^{mp_i}, 0^{mp_i}, 10^{n(p_i+1)-1})$ such that $mp_i = n(p_i+1)$. Using standard facts from number theory, we have that $n \geq p_i$. Hence, $|w_i| \geq 2 + 3 \times p_i(p_i+1)$, that is, $|w_i| = \Theta(|\mathbf{t}_i|^2)$.

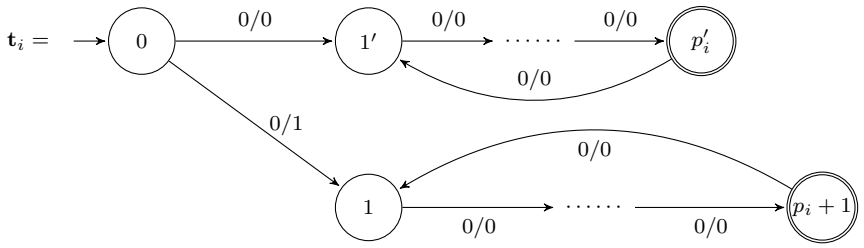


Figure 3: Transducers with quadratic size minimal witnesses of non-functionality.

The following lemma is useful for establishing the correctness of the algorithm `nonFunctionalW`.

Lemma 7. *If a state* (q, q') *has a delay-path value* $((s, s'), (\alpha, \beta, \beta'))$ *then there is a word* h *such that* $\beta = hs$ *and* $\beta' = hs'$*.*

Proof. We use induction based on Definition 4. If the given delay-path value is $((\varepsilon, \varepsilon), (\varepsilon, \varepsilon, \varepsilon))$ the statement is true. Now suppose that there is a transition $((p, p'), (v, x, x'), (q, q'))$ such that the statement is true for state (p, p') (induction hypothesis) and $((s, s'), (\alpha, \beta, \beta'))$ results from a delay-path value $(C, (w, z, z'))$ of (p, p'). As $(s, s') \neq$ ZERO then also $C \neq$ ZERO, so C is of the form (y, y') and one of

the three cases 1–3 of Definition 4 applies. Moreover, by the induction hypothesis on (p, p') we have $z = gy$ and $z' = gy'$, for some word g, hence, $\beta = gyx$ and $\beta' = gy'x'$. Now we consider the three cases. If $y'x' = yxu$ then $(s, s') = (\varepsilon, u)$. Also, for $h = gyx$ we have $\beta = hs$ and $\beta' = hs'$, as required. If $yx = y'x'u$ then $(s, s') = (u, \varepsilon)$ and one works analogously. If $yx = y'x'$ then $(s, s') = (\varepsilon, \varepsilon)$. Also, $\beta = \beta'$ and the statement follows using $h = \beta$. □

Proof. (of Theorem 6) First note that the algorithm returns a triple other than (`None,None,None`) exactly the first time when one of the following occurs (i) a ZERO value for D is computed, or (ii) a value of D other than $(\varepsilon, \varepsilon)$ is computed for a final state, or (iii) a value of D, other than the existing delay value, of a visited state is computed. Thus, the algorithm assigns at most one delay value to each state (q, q'). If the algorithm assigns exactly one delay value to each state and terminates at step 5, then its execution is essentially the same as that of the decision version of the algorithm, except for the fact that in the decision version no path values are computed. Hence, in this case the transducer is functional and the algorithm correctly returns (`None,None,None`) in step 5.

In the sequel we assume that the algorithm terminates in one of the three subcases (a)—(c) of step 4. So let (q, q') be a state at which the algorithm computes some delay value D and path value $R = (\alpha, \beta, \beta')$—see step 4. It is sufficient to show the following statements.

S1 If D is ZERO then $(\alpha w_{qq'}, \beta z_{qq'}, \beta' z'_{qq'})$ is a witness of **t**'s non-functionality.

S2 If (q, q') is final and $D \in \{(\varepsilon, u), (u, \varepsilon)\}$, with u nonempty, then (α, β, β') is a witness of **t**'s non-functionality.

S3 If D is of the form (s, s') and $((s_1, s'_1), (\alpha_1, \beta_1, \beta'_1))$ is the existing delay-path value of (q, q') with $(s_1, s'_1) \neq (s, s')$, then one of the following triples is a witness of **t**'s non-functionality: $(\alpha w_{qq'}, \beta z_{qq'}, \beta' z'_{qq'})$, $(\alpha_1 w_{qq'}, \beta_1 z_{qq'}, \beta'_1 z'_{qq'})$.

For statement S1, by Remark 5, it suffices to show that $\beta z_{qq'} \neq \beta' z'_{qq'}$. First note that D is ZERO exactly when there is a transition $((p, p'), (v, x, x'), (q, q'))$ such that state (p, p') has a delay-path value $((y, y'), (w, z, z'))$ and $yx, y'x'$ are of the form $f\sigma g$ and $f\sigma' g'$, respectively, with σ, σ' being distinct letters, and $\alpha = wv$, $\beta = zx$, $\beta' = z'x'$. By the above lemma, there is a word h such that

$$\beta z_{qq'} = zxz_{qq'} = hyxz_{qq'} = hf\sigma gz_{qq'} \text{ and}$$
$$\beta' z'_{qq'} = z'x'z'_{qq'} = hy'x'z'_{qq'} = hf\sigma' g'z'_{qq'},$$

which implies $\beta z_{qq'} \neq \beta' z'_{qq'}$, as required.

For statement S2, by Remark 5, it suffices to show that $\beta \neq \beta'$. By symmetry, we only consider the case of $D = (\varepsilon, u)$. First note that D is (ε, u) exactly when there is a transition $((p, p'), (v, x, x'), (q, q'))$ such that state (p, p') has a delay-path value $((y, y'), (w, z, z'))$ and $y'x' = yxu$, and $\alpha = wv$, $\beta = zx$, $\beta' = z'x'$. By the above lemma, there is a word h such that $\beta = zx = hyx$ and $\beta' = z'x' = hy'x' = hyxu$, which implies $\beta \neq \beta'$, as required.

For statement S3, we assume that $\beta z_{qq'} = \beta' z'_{qq'}$ and we show that $\beta_1 z_{qq'} \neq \beta_1 z'_{qq'}$. Assume for the sake of contradiction that also $\beta_1 z_{qq'} = \beta_1 z'_{qq'}$. By the above lemma,

there is a word h such that $\beta = hs$, $\beta' = hs'$, $\beta_1 = h_1 s_1$, $\beta'_1 = h_1 s'_1$. Also by the assumptions we get $hsz_{qq'} = hs'z'_{qq'}$ and $h_1 s_1 z_{qq'} = h_1 s'_1 z'_{qq'}$, implying that $sz_{qq'} = s'z'_{qq'}$ and $s_1 z_{qq'} = s'_1 z'_{qq'}$. If $z_{qq'} = z'_{qq'}$ then $s = s' = \varepsilon$ and $s_1 = s'_1 = \varepsilon$, which is impossible as $(s, s') \neq (s_1, s'_1)$. If $z_{qq'}$ is of the form $z_1 z'_{qq'}$ (or vice versa), then we get that $(s, s') = (s_1, s'_1)$, which is again impossible.

Regarding the size of the witness returned, consider again statements S1–S3 above. Then, the size of the witness is $|(x, y, z)| + |(w_{qq'}, z_{qq'}, z'_{qq'})| - 2$, where $(w_{qq'}, z_{qq'}, z'_{qq'})$ could be $(\varepsilon, \varepsilon, \varepsilon)$ and (x, y, z) is a path value of state (q, q'): (α, β, β') or $(\alpha_1, \beta_1, \beta'_1)$. As $(w_{qq'}, z_{qq'}, z'_{qq'})$ is based on a shortest path from (q, q') to a final state of $\mathbf{u}$, we have $|(w_{qq'}, z_{qq'}, z'_{qq'})| < |\mathbf{u}|$. As the algorithm visits each transition of $\mathbf{u}$ at most once, and (x, y, z) is built by concatenating transition labels starting from label $(\varepsilon, \varepsilon, \varepsilon)$, we have that the size of (x, y, z) is bounded by the sum of the sizes of the transitions of $\mathbf{u}$. Hence, the size of the witness is $O(|\mathbf{u}|)$. The claim about the size of the witness follows by the fact that $|\mathbf{u}| = \Theta(|\mathbf{t}|^2)$. □

5. Object Classes Representing Code Properties

In this section we discuss our implementation of objects representing code properties. The set of all code properties is uncountable, but any formal method can describe only countably many properties. Three formal methods are the implicational conditions of [14], where a property is described by a first order formula of a certain type, the regular trajectories of [7], where a property is described by a regular expression over $\{0, 1\}$, and the transducers of [9], where a property is described by a transducer. The formal methods of regular trajectories and transducers are implemented here, as the transducer formal method follows naturally our implementation of transducers, and every regular expression of the regular trajectory formal method can be converted efficiently to a transducer object of the transducer formal method.

Regular trajectory properties [7]. In this formal method a regular expression $\bar{e}$ over $\{0, 1\}$ describes a code property denoted by $\mathcal{P}_{\bar{e}}$. For example, the *infix code* property is described by the regular expression $1^*0^*1^*$, which says that by deleting consecutive symbols at the beginning and/or at the end of an L-word u, one cannot get a different L-word. Equivalently, L is an infix code if no L-word is an infix of another L-word. Note that $1^*0^*1^*$ describes all infix codes over all possible alphabets.

Input-altering transducer properties [9]. A transducer $\mathbf{t}$ is *input-altering* if, for all words w, $w \notin \mathbf{t}(w)$. In this formal method such a transducer $\mathbf{t}$ describes the code property $\mathcal{P}^{\mathrm{al}}_{\mathbf{t}}$ consisting of all languages L over the input alphabet of $\mathbf{t}$ such that

$$\mathbf{t}(L) \cap L = \emptyset. \tag{1}$$

The transducer in Example 3 is input-altering and describes the suffix code property over the alphabet `{a, b}`: L is a suffix code if no L-word is a proper suffix of an L-word. Similarly, we can define the infix code property by making another transducer that, on input w, returns any proper infix of w. We note that, for every regular

expression $\bar{e}$ over $\{0,1\}$ and alphabet Σ, one can construct in linear time an input-altering transducer $\mathbf{t}$ with input alphabet Σ such that $\mathcal{P}_{\bar{e}} = \mathcal{P}^{al}_{\mathbf{t}}$ [9]. Thus, every regular trajectory property is an input-altering transducer property.

Error-detecting properties via input-preserving transducers [9,16]. A transducer $\mathbf{t}$ is *input-preserving* if, for all words w in the domain of $\mathrm{R}(\mathbf{t})$, $w \in \mathbf{t}(w)$. Such a transducer $\mathbf{t}$ is also called a *channel transducer*, in the sense that an input message w can be transmitted via $\mathbf{t}$ and the output can be, either w (no transmission error), or a word other than w (error). In this formal method the transducer $\mathbf{t}$ describes the *error-detecting for* $\mathbf{t}$ property $\mathcal{P}^{ed}_{\mathbf{t}}$ consisting of all languages L over the input alphabet of $\mathbf{t}$ such that

$$\mathbf{t}(w) \cap (L - w) = \emptyset, \quad \text{for all words } w \in L. \tag{2}$$

If L is error-detecting ***for*** $\mathbf{t}$, then $\mathbf{t}$ cannot turn an L-word into a different L-word. We note that, for every input-altering transducer $\mathbf{t}$, one can make in linear time a channel transducer $\mathbf{t}'$ such that $\mathcal{P}^{al}_{\mathbf{t}} = \mathcal{P}^{ed}_{\mathbf{t}'}$ [9]. Thus, every input-altering transducer property is an error-detecting property.

Example 8. Consider the property *1-substitution error-detecting code* over `{a, b}`, where error means the substitution of one symbol by another symbol. A classic characterization is that, L is such a code if and only if the Hamming distance between any two different words in L is at least 2 [11]. The following channel transducer defines this property—see also Fig 4. The transducer will substitute at most one symbol of the input word with another symbol.

```
s1   = '@Transducer 0 1 * 0\n'\
       '0 a a 0\n0 b b 0\n0 b a 1\n'\
       '0 a b 1\n1 a a 1\n1 b b 1\n'
t1 = fio.readOneFromString(s1)
```

□

The transducer approach to defining error-detecting code properties is very powerful, as it allows one to model insertion and deletion errors, in addition to substitution errors—see Fig 4. Codes for such errors are actively under investigation [25].

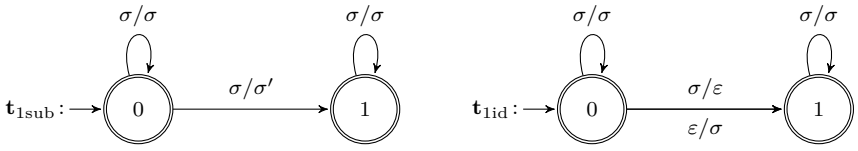


Figure 4: On input x, the transducer $\mathbf{t}_{1\mathrm{sub}}$ outputs either x, or any word that results by substituting exactly one symbol in x. On input x, the transducer $\mathbf{t}_{1\mathrm{id}}$ outputs either x, or any word that results by deleting, or inserting, exactly one symbol in x. Note: The use of labels on arrows is explained in Fig. 1.

Remark 9. All input-altering and error-detecting properties are 3-independences.

5.1. Implementation in FAdo.

We have defined the Python classes `TrajProp, IATProp, ErrDetectProp` corresponding to the three types of properties discussed above. Properties of these types are described, respectively, by regular trajectory expressions, input-altering transducers, input-preserving transducers. An object `p` of the class `IATProp` is defined via some SFT `t` and represents a particular code property, that is, the set of languages satisfying Eq. (1). The class `ErrDetectProp` is a superclass of the others. These classes and all related methods are in the module `codes.py`. We have defined a set of what we call *build functions* as a user interface for creating code property objects. These build functions are shown next with examples.

Example 10. Consider the strings `s` and `s1` (see Examples 3 and 8) containing, respectively, the proper suffixes transducer and the transducer permitting up to 1 substitution error. The following object definitions are possible in FAdo.

```
import FAdo.codes as codes
icp = codes.buildTrajPropS('1*0*1*', {'a', 'b'})
scp = codes.buildIATPropS(s)
s1dp = codes.buildErrorDetectPropS(s1)
pcp = codes.buildPrefixProperty({'a', 'b'})
icp2 = codes.buildInfixProperty({'a', 'b'})
```

The object `icp` represents the infix code property over the alphabet `{a, b}` and is defined via the trajectory expression `1*0*1*`. In the next two statements, `scp, s1dp` represent, respectively, the suffix code property and the 1-substitution error-detecting property. In the last two statements the objects `pcp` and `icp2` represent the prefix code and infix code properties, respectively (see 'Fixed properties' below). □

Fixed properties. For some well-known properties in the theory of codes we have created specific classes and, therefore, FAdo users need not write transducers, or trajectory regular expressions, for creating these properties. These properties are prefix codes, suffix codes, infix codes, outfix codes, and hypercodes. As before, users need only to know about the `build`-interfaces for creating objects of these classes.

Combining code properties. In practice we need to talk about languages satisfying more than one property. For example, most of the 1-substitution error-detecting codes used in practice are infix codes. We have defined the operation `&` between any two error-detecting properties independently of how they were created. This operation returns an object representing the class of all languages satisfying both properties. This object is constructed via the transducer that results by taking the union of the two transducers describing the two properties—see Rational Operations in Section 3.

Example 11. Using the properties `icp, s1dp` created in Example 10, we can create the conjunction `p1` of these properties, and using the properties `pcp, scp` we can create their conjunction `bcp` which is known as the *bifix code property*.

```
p1 = icp & s1dp
bcp = pcp & scp
```

The object `p1` represents the property $\mathcal{P}_{\bar{e}} \cap \mathcal{P}^{ed}_{\mathtt{s1}}$, where $\bar{e} =$ `1*0*1*`. It is of type `ErrDetectProp`. If, however, the two properties involved are input-altering then our implementation makes sure that the object returned is also of type input-altering—this is the case for `bcp`. This is important as the satisfaction problem for input-altering transducer properties can be solved more efficiently than the satisfaction problem for error-detecting properties. □

5.2. Aspects of Code Hierarchy Implementation

As stated above, our top Python superclass is `ErrDetectProp`. When viewed as a set of (potential) objects, this class implements the set of properties

$$\mathcal{P}^{ed} = \{\mathcal{P}^{ed}_{\mathbf{t}} \mid \mathbf{t} \text{ is an input-preserving transducer}\}. \tag{3}$$

For any `ErrDetectProp` object `p`, let us denote by [p] the property in $\mathcal{P}^{ed}$ represented by `p`. If `p` and `q` are any `ErrDetectProp` objects such that [p] $\subseteq$ [q] and we know that a language satisfies [p] then it follows logically that the language also satisfies [q] and, therefore, one does not need to execute the method `q.notSatsfiesW` on the automaton accepting that language. Similarly, as [p&q] = [p] the method invocation `p&q` should simply return `p`. It is therefore desirable to have a method '$\leq$' such that if `p` $\leq$ `q` returns true then [p] $\subseteq$ [q]. In fact, for `ErrDetectProp` objects, we have implemented methods for '&' and '$\leq$' in a way that the triple (`ErrDetectProp`, &, $\leq$) constitutes a syntactic hierarchy (see further below) which can be used to simulate all properties in (3). In practice this means that '&' simulates intersection between properties and '$\leq$' simulates subset relationship between two properties such that the following desirable statements hold true, for any `ErrDetectProp` objects `p, q`

`p & p` returns `p`

`p` $\leq$ `q` if and only if `p & q` returns `p`

We note that the syntactic simulation of the properties in Eq. (3) is not complete (in fact it cannot be complete): for any `ErrDetectProp` objects `p, q` defined via transducers $\mathbf{t}$ and $\mathbf{s}$ with $\mathcal{P}^{ed}_{\mathbf{t}} \subseteq \mathcal{P}^{ed}_{\mathbf{s}}$ it does not always hold that `p` $\leq$ `q`. On the other hand, our implementation of the set of the five fixed properties constitutes a complete simulation of these properties, when the same alphabet is used. This implies, for instance, that `pcp & icp2` returns `icp2`, where we have used the notation of Example 10. Our implementation associates to each object `p` of type `ErrDetectProp` a nonempty set p.ID of names. If `p` is a fixed property object, p.ID has one hardcoded name. If `p` is built from a transducer $\mathbf{t}$, p.ID has one name, the name of $\mathbf{t}$—this name is based on a string description of $\mathbf{t}$. If `p` = `q&r`, then p.ID is the union of q.ID and r.ID minus any fixed property name N for which another fixed property name M exists in the union such that the M-property is contained in the N-property.

Next we give a mathematical definition of what it means to simulate a set of code properties $\mathcal{Q} = \{\mathcal{Q}_j\}_{j\in J}$ via a syntactic hierarchy $(G, \&, \leq)$, which can ultimately be

implemented (as is the case here) in a standard programming language. The idea is that each $g \in G$ represents a property $[g] = \mathcal{Q}_j$, for some index j, and G is the set of generators of the semigroup $(\langle G\rangle, \&)$ whose operation '&' simulates the process of combining properties in $\mathcal{Q}$, that is $[x\&y] = [x]\cap[y]$, and the partial order '$\leq$' simulates subset relation between properties, that is $x \leq y$ implies $[x] \subseteq [y]$, for all $x, y \in \langle G\rangle$. Theorem 15 says that there is a simulation of the set $\mathcal{P}^{ed}$ in Eq. (3). Theorem 16 says that there can be no *complete* simulation of that set of properties, that is, a simulation such that $[x] \subseteq [y]$ implies $x \leq y$, for all $x, y \in \langle G\rangle$.

Definition 12. A *syntactic hierarchy* is a triple $(G, \&, \leq)$ such that G is a nonempty set and

(I) $(\langle G\rangle, \&)$ is the commutative semigroup generated by G with computable operation '&'.

(II) $(\langle G\rangle, \leq)$ is a decidable partial order (reflexive, transitive, antisymmetric).

(III) For all $x, y \in \langle G\rangle$, $x \leq y$ implies $x\&y = x$.

(IV) For all $x, y \in \langle G\rangle$, $x\&y \leq x$.

Next we list a few properties of the operation '&' and the order '$\leq$'.

Lemma 13. *The following statements hold true, for all $x, y, z \in \langle G\rangle$,*

(I) *$x \leq x$ and $x\&x = x$*

(II) *$x \leq y$ if and only if $x = y\&z$ for some $z \in \langle G\rangle$.*

(III) *$x = x\&y$ if and only if $x \leq y$.*

(IV) *If $x \leq y$ and $x \leq z$ then $x \leq y\&z$.*

(V) *If $x = g_1\& \cdots \&g_n$, for some $g_1, \ldots, g_n \in G$, with all g_i's distinct and $n \geq 2$, then $x < g_1$ or $x < g_2$, and hence x is not maximal.*

(VI) *x is maximal if and only if x is prime (meaning, $x = u\&v$ implies $x = u = v$).*

Proof. The proof of correctness is based on Definition 12 using standard logical arguments. We prove only the second and fourth statements. The 'if' part of the second statement follows from the fourth part of Definition 12, and the 'only if' part follows from the third part of that definition. For the fourth statement, using the fact that $x\&(y\&z) \leq y\&z$, it is sufficient to show that $x = x\&(y\&z)$. This follows when we note that $x \leq y$ implies $x = x\&y$ and $x \leq z$ implies $x = x\&z$. □

Definition 14. Let $\mathcal{Q} = \{\mathcal{Q}_j\}_{j\in J}$ be a set of properties. A *(syntactic) simulation* of $\mathcal{Q}$ is a quintuple $(G, \&, \leq, [\,], \varphi)$ such that $(G, \&, \leq)$ is a syntactic hierarchy and

(I) $[\,]$ is a surjective mapping of $\langle G\rangle$ onto $\mathcal{Q}$;

(II) for all $x, y \in \langle G\rangle$, $x \leq y$ implies $[x] \subseteq [y]$;

(III) for all $x, y \in \langle G\rangle$, $[x\&y] = [x] \cap [y]$;

(IV) φ is a computable function of J into $\langle G\rangle$ such that $[\varphi(j)] = \mathcal{Q}_j$.

The simulation is called *complete* if, $[x] \subseteq [y]$ implies $x \leq y$, for all x, y.

Theorem 15. *There is a simulation of the set of properties $\mathcal{P}^{\mathrm{ed}}$.*

Proof. Let $G = \{\{\mathbf{t}\} \mid \mathbf{t} \text{ is an input-preserving transducer}\}$, and let $\mathbf{T}$ be the set consisting of all finite sets of transducers. For any $T_1, T_2 \in \mathbf{T}$, we define

$$T_1 \& T_2 = T_1 \cup T_2 \quad \text{and} \quad T_1 \leq T_2, \text{ if } T_2 \subseteq T_1.$$

The above definitions imply that $\langle G \rangle$ consists of all T, where T is a finite nonempty set of input-preserving transducers, and that indeed $(\langle G \rangle, \&)$ is a commutative semigroup and $(\langle G \rangle, \leq)$ is a partial order. Moreover one verifies that the last two requirements of Definition 12 are satisfied. Thus $(G, \&, \leq)$ is a syntactic hierarchy.

Next we use the syntactic hierarchy $(G, \&, \leq)$ to define the required simulation. First, let $\varphi(\mathbf{t}) = \{\mathbf{t}\}$, for any input-preserving transducer $\mathbf{t}$. Then, let $[T] = \mathcal{P}^{ed}_{\vee T}$, which is equal to $\bigcap_{\mathbf{t} \in T} \mathcal{P}^{\mathrm{ed}}_{\mathbf{t}}$. One verifies that the requirements of Definition 14 are satisfied. □

The proof of the next result is based on the undecidability of the *Post Correspondence Problem (PCP)* and uses methods for establishing the undecidability of basic transducer problems [4].

Theorem 16. *There is no complete simulation of the set of properties* $\mathcal{P}^{\mathrm{ed}}$.

Before we present the proof, we establish a few necessary auxiliary results.

Lemma 17. *For any input-preserving transducers* $\mathbf{t}, \mathbf{s}$ *we have that* $\mathcal{P}^{\mathrm{ed}}_{\mathbf{t}} \subseteq \mathcal{P}^{\mathrm{ed}}_{\mathbf{s}}$ *if and only if* $\mathrm{R}(\mathbf{s} \vee \mathbf{s}^{-1}) \subseteq \mathrm{R}(\mathbf{t} \vee \mathbf{t}^{-1})$.

Proof. First note that Eq. (2) is equivalent to "$(w, v) \in \mathrm{R}(\mathbf{t})$ implies $w = v$, for all $v, w \in L$." This implies that, for all input-preserving transducers $\mathbf{t}, \mathbf{s}$, we have $\mathcal{P}^{\mathrm{ed}}_{\mathbf{s}} = \mathcal{P}^{ed}_{\mathbf{s}^{-1}}$ and "$\mathrm{R}(\mathbf{s}) \subseteq \mathrm{R}(\mathbf{t})$ implies $\mathcal{P}^{ed}_{\mathbf{t}} \subseteq \mathcal{P}^{ed}_{\mathbf{s}}$" and $\mathcal{P}^{ed}_{\mathbf{s}} = \mathcal{P}^{ed}_{\mathbf{s} \vee \mathbf{s}^{-1}}$. The statement to prove follows from these observations using standard set theoretic arguments. □

Lemma 18. *The following problem is undecidable.*
Input: input-preserving transducers $\mathbf{t}, \mathbf{s}$
Answer: whether $\mathrm{R}(\mathbf{s} \vee \mathbf{s}^{-1}) \subseteq \mathrm{R}(\mathbf{t} \vee \mathbf{t}^{-1})$

Proof. We reduce PCP to the given problem. Consider any instance $\big((u_i)_{i=1}^p, (v_i)_{i=1}^p\big)$ of PCP which is a pair of sequences of p nonempty words over some alphabet B, for some positive integer p. As in [4], we have that the given instance is a YES instance if and only if $U^+ \cap V^+ \neq \emptyset$, where

$$U = \{(ab^i, u_i) \mid i = 1, \ldots, p\} \text{ and } V = \{(ab^i, v_i) \mid i = 1, \ldots, p\},$$

and we make no assumption about the intersection of the alphabets B and $\{a, b\}$. Here we define the following objects

$$C = \{ab, ab^2, \ldots, ab^p\} \tag{4}$$

$$\mathrm{diag}(L) = \{(x, x) \mid x \in L\}, \text{ for any language } L \tag{5}$$

$$D = \mathrm{diag}(C^+) \cup \mathrm{diag}(aaB^+) \tag{6}$$

$$X = (\varepsilon, aa)U^+ \cup D \tag{7}$$

$$Y = ((C^+ \times aaB^+) - (\varepsilon, aa)V^+) \cup D \tag{8}$$

Let $\mathbf{t}$ and $\mathbf{s}$ be any transducers realizing, respectively, X and Y. We shall prove the following three claims, which establish the required problem reduction.

C1: X and Y are rational relations

C2: $\mathbf{t}$ and $\mathbf{s}$ are input preserving

C3: $U^+ \cap V^+ \neq \emptyset$ if and only if $(X \cup X^{-1}) \subseteq (Y \cup Y^{-1})$

Claim C1: As C^+ and aaB^+ are regular languages, D is a rational relation. Also, [4] shows that U^+ and $((C^+ \times B^+) - V^+)$ are rational relations. It follows then that X is a rational relation. Similarly, rational is also the relation

$$(\varepsilon, aa)((C^+ \times B^+) - V^+) = ((C^+ \times aaB^+) - (\varepsilon, aa)V^+),$$

which implies that Y is rational as well.

Claim C2: From the previous claim there are transducers (in fact effectively constructible) $\mathbf{t}$ and $\mathbf{s}$ realizing X and Y, respectively. Note that the domains of both transducers are equal to $C^+ \cup aaB^+$. The fact that $(x, x) \in \mathrm{R}(\mathbf{t}) \cap \mathrm{R}(\mathbf{s})$ for all $x \in C^+ \cup aaB^+$ implies that both transducers are indeed input-preserving.

Claim C3: First note that $X^{-1} \subseteq Y^{-1}$ if and only if $X \subseteq Y$ (for any relations X and Y). Then, $C^+ \cap aaB^+ = \emptyset$ implies that $(\varepsilon, aa)U^+$ is disjoint from the sets $((\varepsilon, aa)U^+)^{-1}$, D, $((C^+ \times aaB^+) - (\varepsilon, aa)V^+)^{-1}$ and similarly $((C^+ \times aaB^+) - (\varepsilon, aa)V^+)$ is disjoint from the same sets. The above observations imply that

$(X \cup X^{-1}) \subseteq (Y \cup Y^{-1})$ if and only if

$(\varepsilon, aa)U^+ \subseteq ((C^+ \times aaB^+) - (\varepsilon, aa)V^+)$ if and only if

$(\varepsilon, aa)U^+ \cap (\{a, b\}^* \times B^*) - ((C^+ \times aaB^+) - (\varepsilon, aa)V^+) = \emptyset$ if and only if

$(\varepsilon, aa)U^+ \cap (\varepsilon, aa)V^+ = \emptyset$ if and only if $U^+ \cap V^+ = \emptyset$, as required. □

Proof. (of Theorem 16.) For the sake of contradiction, assume there is a complete simulation $(G, \&, \leq, [\,], \varphi)$ of $\mathcal{P}^{\mathrm{ed}}$. Then, $[x] \subseteq [y]$ implies $x \leq y$, for all $x, y \in \langle G \rangle$. We get a contradiction by showing that the problem in Lemma 18 is decided as follows.

1. Let $x = \varphi(\mathbf{t})$
2. Let $y = \varphi(\mathbf{s})$
3. if $y \leq x$: return YES
4. else: return NO

The correctness of the 'if' clause is established as follows: $y \leq x$ implies $[y] \subseteq [x]$, which implies $\mathcal{P}^{ed}_{\mathbf{t}} \subseteq \mathcal{P}^{ed}_{\mathbf{s}}$, and then $\mathrm{R}(\mathbf{s} \vee \mathbf{s}^{-1}) \subseteq \mathrm{R}(\mathbf{t} \vee \mathbf{t}^{-1})$, as required. The correctness of the 'else' clause is established as follows: first note $y \not\leq x$. We need to show $\mathrm{R}(\mathbf{s} \vee \mathbf{s}^{-1}) \not\subseteq \mathrm{R}(\mathbf{t} \vee \mathbf{t}^{-1})$. For the sake of contradiction, assume the opposite. Then, $\mathcal{P}^{ed}_{\mathbf{t}} \subseteq \mathcal{P}^{ed}_{\mathbf{s}}$, which implies $[y] \subseteq [x]$, and then (by completeness) $y \leq x$, which is a contradiction. □

6. Methods of Code Property Objects

In the context of the research on code properties, we consider the following three algorithmic problems as fundamental. *Satisfaction problem*: Given the description of a code property and the description of a language, decide whether the language satisfies the property. In the *witness version* of this problem, a negative answer is also accompanied by an appropriate list of words showing how the property is violated. *Maximality problem*: Given the description of a code property and the description of a language L, decide whether the language is maximal with respect to the property. In fact we allow the more general problem, where the input includes also the description of a second language M and the question is whether there is no word $w \in M \setminus L$ such that $L \cup w$ satisfies the property. The default case is when $M = \Sigma^*$. In the *witness version* of this problem, a negative answer is also accompanied by any word w that can be added to the language L. *Construction problem*: Given the description of a code property and two positive integers n and ℓ, construct a language that satisfies the property and contains n words of length ℓ (if possible).

In the above problems, and in the rest of the section, it is assumed that the code property is implemented as an object `p` via a transducer $\mathbf{t}$ (whether input-altering or input-preserving). In the first two problems, the language is given via an NFA $\mathbf{a}$. In the maximality problem, the second language M is given via an NFA $\mathbf{b}$. Next we discuss the implementation of methods for the satisfaction and maximality problems. Aspects of the construction problem are discussed in [17].

Methods `p.maximalP(a, b)`, `p.notMaximalW(a, b)` The maximality problem is decidable but PSPACE-hard [9]. In particular, for the case of both input-altering transducer and error-detecting properties, the decision algorithm is very simple: the language $\mathrm{L}(\mathbf{a})$ is `p`-maximal (within $\mathrm{L}(\mathbf{b})$) if and only if

$$\mathrm{L}(\mathbf{b}) \setminus (\mathrm{L}(\mathbf{a}) \cup \mathbf{t}(\mathbf{a}) \cup \mathbf{t}^{-1}(\mathbf{a})) = \emptyset. \tag{9}$$

The above test is implemented in the method `p.maximalP(a, b)`, using standard NFA methods and the transducer methods `t.inverse()` and `t.runOnNFA(a)`. In fact, [9], any word belonging to the set on the left-hand side of Eq. (9) can be added to $\mathrm{L}(\mathbf{a})$ without violating the property (that word can serve as a witness of the non-maximality of $\mathrm{L}(\mathbf{a})$). This is implemented in `p.notMaximalW(a, b)`. If no such word exists, the method returns `None`.

Methods `p.satisfiesP(a)`, `p.notSatisfiesW(a)` The satisfaction problem is decidable in time $O(|\mathbf{t}||\mathbf{a}|^2)$, if `p` is an input-altering transducer property—this follows from Eq. (1). If `p` is an error-detecting property, the transducer $\mathbf{t}$ is input-preserving and Eq. (2) is decided via a transducer functionality test, [16]. The witness version of the method `p.satisfiesP(a)` returns either (`None`, `None`), or a pair of ***different*** words $u, v \in \mathrm{L}(\mathtt{a})$ violating the property, that is, $v \in \mathtt{t}(u)$ or $u \in \mathtt{t}(v)$. In the latter case, the pair (u, v) is called a *witness of the non-satisfaction of* `p` *by* the language $\mathrm{L}(\mathtt{a})$. Next we discuss how to accomplish this.

Case 1: For input-altering transducer properties, we have the code

```
return t.inIntersection(a).outIntersection(a).nonEmptyW()
```

Recall from Section 3, the above returns (if possible) a witness for the nonemptiness of the transducer `t` when the input and output parts of `t` are intersected by the language L(`a`). This witness corresponds to the emptiness test in Eq. (1), as required.

Case 2: For error-detecting properties `p` , the defining transducer is a channel (input-preserving) and, therefore, we use the method `nonFunctionalW()` instead of `nonEmptyW()`. More specifically, we use the code

```
u, v, w = t.inIntersection(a).outIntersection(a).nonFunctionalW()
if u == v:  return u, w
else:  return u, v
```

If `nonFunctionalW()` returns a triple of words (u, v, w) then, by Proposition 6 and the definitions of the `in/out` intersection methods, we have that $v \neq w$, $v \in \mathtt{t}(u)$, $w \in \mathtt{t}(u)$ and all three words are in L(`a`). This implies that at least one of $u \neq v$ and $u \neq w$ must be true and, therefore, the returned value is the pair (u, v) or (u, w). Moreover, the returned pair indeed violates the property. Conversely, if `nonFunctionalW()` returns a triple of `None`s then the constructed transducer is not functional. Then L(`a`) must satisfy the property, otherwise any different words $v, w \in$ L(`a`) violating the property could be used to make the triple (v, v, w), or (w, w, v), which would serve as a witness of the non-functionality of the transducer.

The above discussion establishes the following consequence of Proposition 6 and of the definitions of product constructions in Section 3.

Proposition 19. *The algorithms implemented in the two forms (input-altering transducer, error-detecting) of the method* `p.notSatisfiesW(a)` *correctly return a witness of the non-satisfaction of the property* `p` *by the language* L(`a`).

Example 20. The following Python interaction shows that the language a^*b is a prefix and 1-error-detecting code. Recall from previous examples that the strings `st, s1` contain, respectively, the descriptions of an NFA accepting a^*b, and a channel transducer that performs up to one substitution error when given an input word.

```
>>> a = fio.readOneFromString(st)
>>> pcp = codes.buildPrefixProperty({'a','b'})
>>> s1dp = codes.buildErrDetectPropS(s1)
>>> p2 = pcp & s1dp
>>> p2.notSatisfiesW(a)
(None, None)
```

□

7. Uniquely Decodable/Decipherable Codes

The property of unique decodability or decipherability, *UD code property* for short, is probably the first historically property of interest from the points of view of both information theory [31] as well as formal language theory [24, 32]. A language L is said to be a UD code if, for any two sequences $(u_i)_1^n$ and $(v_j)_1^m$ of L-words such that

$u_1 \cdots u_n = v_1 \cdots v_m$, we have that $n = m$ and $u_i = v_i$ for all i; that is, every word in L^* can be decomposed in exactly one way as a sequence of L-words. In this section we discuss our implementation of the satisfaction and maximality methods for the UD code property—this property is not an n-independence for any $n \in \mathbb{N}$, [15], so it cannot be described by any transducer.

The method `p.notSatisfiesW(a)` The satisfaction problem for this property was discussed first in the well known paper [31], where the authors produce a necessary and sufficient condition for a finite language to be a UD code—some feel that that condition does not clearly give an algorithm, as for instance the papers [19, 21]. Over the years people have established faster algorithms and generalized the problem to the case of regular languages. The asymptotically fastest algorithms for regular languages appear to be the ones in [12, 23], both of quadratic time complexity. Our implementation follows the algorithm in [12], which makes use of a certain transducer functionality test. Again we have enhanced that algorithm to produce a *witness of non-satisfaction* which, given an NFA object `a`, it either returns (`None`, `None`) if L(`a`) is a UD code, or a pair of two different lists of L(`a`)-words such that the concatenation of the words in each list produces the same word (if L(`a`) is not a UD code).

Example 21. The following Python interaction produces a witness of the non-satisfaction of the UD code property by the language $\{ab, abba, bab\}$.

```
>>> L = fl.FL(['ab', 'abba', 'bab'])
>>> a = L.toNFA()
>>> p = codes.buildUDCodeProp(a.Sigma)
>>> p.notSatisfiesW(a)
(['ab', 'bab', 'abba', 'bab'], ['abba', 'bab', 'bab', 'ab'])
```

The two word lists are different, but the concatenation of the words in each list produces the same word. □

The method `p.maximalP(a)` This method is based on the theorem of Schützenberger [5] that a regular language L is a UD code if and only if the set of all infixes of L^* is equal to Σ^*. Using the tools implemented in FAdo this test can be performed as follows, where `t` is a transducer that returns any proper infix of the input.

```
b = a.star()
return (~(t.runOnNFA(b))).emptyP()
```

The first statement returns an NFA accepting L(`a`)*, and the last statement returns whether there is no word in the complement of all infixes of L(**a**)*.

8. New version of LaSer and Program Generation

The first version of LaSer [9] was a self-contained set of C++ automaton and transducer methods as well as a set of Python and HTML documents with the following functionality: a user uploads a file containing an automaton in Grail format and a file containing either a trajectory automaton, or an input altering-transducer, and LaSer responds with an answer to the witness version of the satisfaction problem. The new

version is based on the FAdo set of automaton and transducer methods and allows users to request a response about the witness versions of the satisfaction and maximality problems for input-altering transducer, error-detecting and error-correcting properties. We call the above type of functionality, where LaSer computes and returns the answer, the *online service* of LaSer. Another feature of the new version of LaSer, which we believe to be original in the community of software on automata and formal languages, is the *program generation service*. This is the capability to generate a self-contained Python program that is downloaded on the users side and executed on their machine returning thus the desired answer. This feature is useful as the execution of certain algorithms, even of polynomial time complexity, can be too time consuming to do on the server side.

9. Some Test Runs of the Implemented Methods

We present a few test runs of the method `p.notSatisfiesW(a)` and the method `p.notMaximalW(a, b)`, where the property `p` is input-altering or error-detecting.

In Fig. 5, we consider three input-altering transducers describing infix codes, suffix codes, prefix codes—see Fig. 1—and the regular languages $L_n = 01^*0+1^n$, for various values of $n \in \mathbb{N}$. For each n, the language L_n is not an infix code, but it is both a prefix and suffix code. We ran satisfaction and maximality tests for several values of $n \in [10, 1000]$ and step 10. Maximality testing is relative to Σ^*.

In Fig, 6, we consider the input-preserving transducers $\mathbf{t}_{k\text{-sub}}$ describing the k-substitution error-detecting properties, for various $k \in \mathbb{N}$ (see Fig. 4 for $k = 1$) . We also consider the even parity code E_n = all binary words wb with $|w| = n - 1$ and b being the bit that makes the total number of 1's in wb even. It is well-known that E_n is a maximal 1-substitution error-detecting code, where maximality is relative to Σ^n. On the other hand, E_n is not k-substitution error-detecting for any $k \geq 2$.

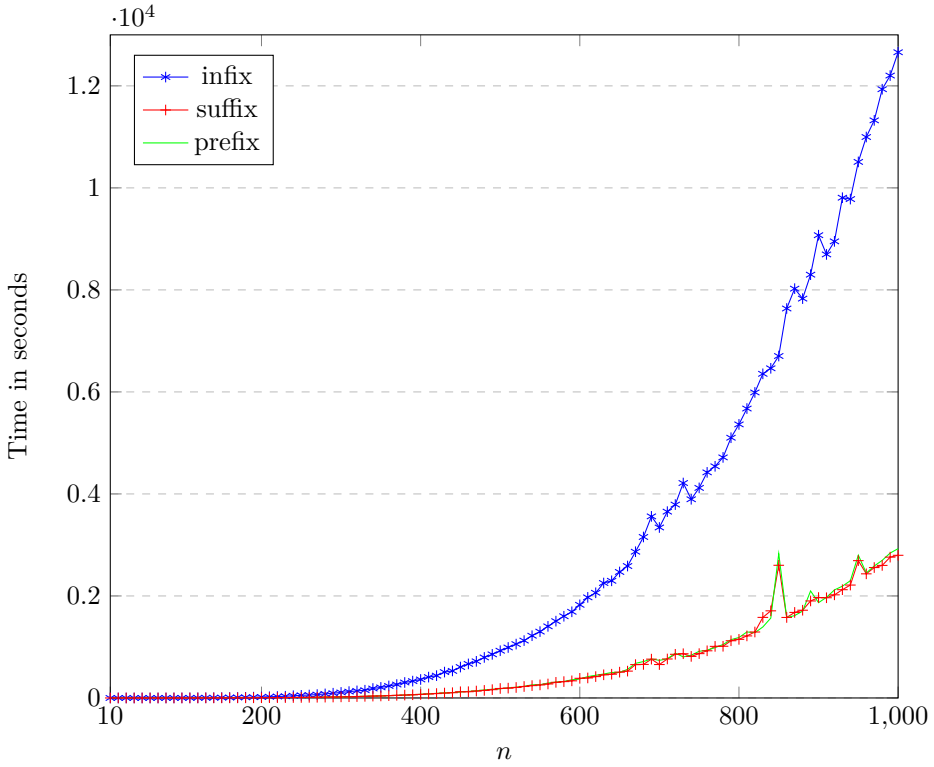
All experiments were performed with PyPy, which implements the Python language version 2.7.13 [26], on a Mac Pro 2.5 GHz Intel Core i7, with 16 GB of memory. A few related experiments have been done in [17]. Our experiments show that the implemented methods can handle automata of moderate size. More thorough testing is needed to find out average empirical running times.

10. Concluding Remarks

We have presented a simple to use set of methods that allow one to define and combine many natural code properties, and obtain answers to the satisfaction and maximality problems. This capability relies on our implementation of basic transducer methods, including our witness version of the non-functionality transducer method, in the FAdo package. We have also produced a new version of LaSer that allows users to inquire about error-detecting and -correcting properties, and to generate programs that can be executed and provide answers at the user's site.

There are a few important directions for future research. First, the existing format of transducer objects is not always efficient when it comes to describing code properties. For example, the transducer `t1` in Example 8 consists of 6 transitions. If the

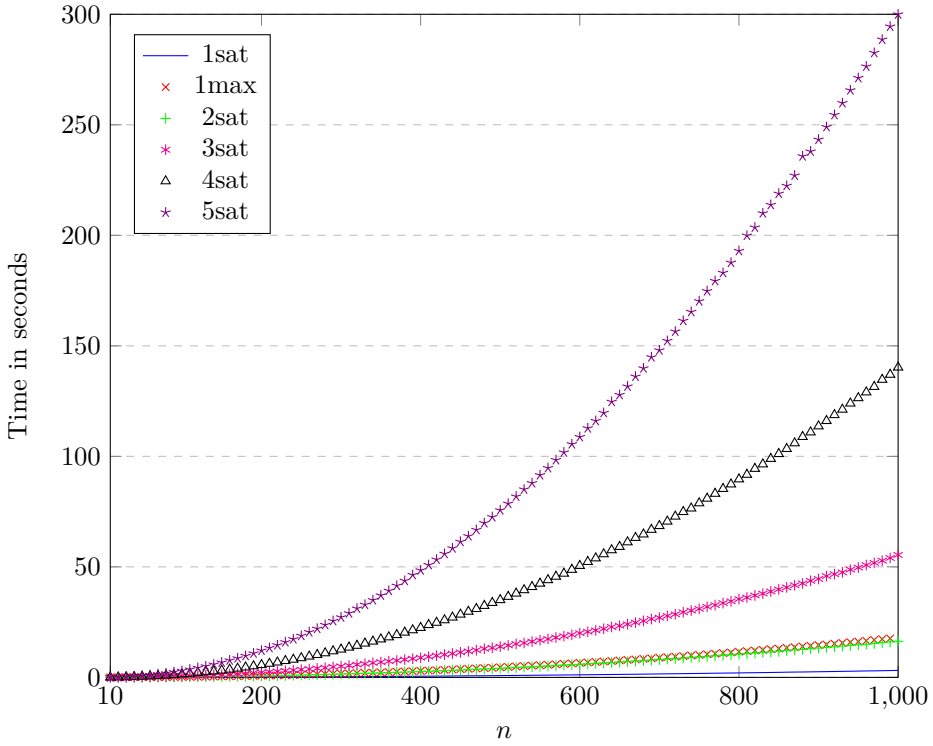
Figure 5: Input-altering tests for L_n. Prefix maximality tests runned as faster as suffix satisfaction tests. A fitting of the curves show a quadratic behaviour in all cases.



alphabet is of size s, then that transducer would require $s + s(s-1) + s = s^2 + s$ transitions. A symbolic notation for transitions would be more compact and can possibly be employed by modifying the appropriate transducer methods. We note that a general method for symbolic transducers is already investigated in [37].

Formal methods for defining code properties need to be better understood or new ones need to be developed, with the aim of ultimately implementing these properties and answering efficiently related problems. These formal methods should allow one to express properties that cannot be expressed with transducer methods, such as the comma-free code property [33]. The method of [14] is quite expressive, using certain first order formulae to describe properties. It could perhaps be further worked out in a way that some of these formulae can be mapped to finite-state machine objects that are, or can be, implemented in available formal language packages like FAdo. We also note that if the method is too expressive then even the satisfaction problem could become undecidable—e.g., the method of multiple sets of trajectories [8].

Figure 6: Error-detecting tests for E_n. The running times for testing maximality for $k = 1$ almost coincide with the ones for the test of satisfaction for $k = 2$.



References

[1] Cyril Allauzen and Mehryar Mohri. Efficient algorithms for testing the twins property. *Journal of Automata, Languages and Combinatorics*, 8(2):117–144, 2003.

[2] André Almeida, Marco Almeida, José Alves, Nelma Moreira, and Rogério Reis. FAdo and GUItar: Tools for automata manipulation and visualization. In Sebastian Maneth, editor, *Proceedings of CIAA 2009, Sydney, Australia*, volume 5642 of *Lecture Notes in Computer Science*, pages 65–74, 2009.

[3] Marie-Pierre Béal, Olivier Carton, Christophe Prieur, and Jacques Sakarovitch. Squaring transducers: An efficient procedure for deciding functionality and sequentiality. *Theoretical Computer Science*, 292(1):45–63, 2003.

[4] Jean Berstel. *Transductions and Context-Free Languages*. B.G. Teubner, Stuttgart, 1979.

[5] Jean Berstel, Dominique Perrin, and Christophe Reutenauer. *Codes and Automata*. Cambridge University Press, 2009.

[6] Akim Demaille, Alexandre Duret-Lutz, Sylvain Lombardy, and Jacques Sakarovitch. Implementation concepts in vaucanson 2. In Stavros Konstantinidis, editor, *Proceedings of CIAA 2013*, volume 7982 of *Lecture Notes in Computer Science*, pages 122–133, 2013.

[7] Michael Domaratzki. Trajectory-based codes. *Acta Informatica*, 40:491–527, 2004.

[8] Michael Domaratzki and Kai Salomaa. Codes defined by multiple sets of trajectories. *Theoretical Computer Science*, 366:182–193, 2006.

[9] Krystian Dudzinski and Stavros Konstantinidis. Formal descriptions of code properties: decidability, complexity, implementation. *International Journal of Foundations of Computer Science*, 23:1:67–85, 2012.

[10] FAdo. Tools for formal languages manipulation. URL address: `http://fado.dcc.fc.up.pt/` Accessed in December of 2017.

[11] Richard W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, 1950.

[12] Tom Head and Andreas Weber. Deciding code related properties by means of finite transducers. In R. Capocelli, A. de Santis, and U. Vaccaro, editors, *Sequences II, Methods in Communication, Security, and Computer Science*, pages 260–272. Springer-Verlag, 1993.

[13] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[14] Helmut Jürgensen. Syntactic monoids of codes. *Acta Cybernetica*, 14:117–133, 1999.

[15] Helmut Jürgensen and Stavros Konstantinidis. Codes. In Rozenberg and Salomaa [29], pages 511–607.

[16] Stavros Konstantinidis. Transducers and the properties of error-detection, error-correction and finite-delay decodability. *Journal of Universal Computer Science*, 8:278–291, 2002.

[17] Stavros Konstantinidis, Nelma Moreira, and Rogério Reis. Generating error control codes with automata and transducers. In Henning Bordihn, Rudolf Freund, Benedek Nagy, and György Vaszil, editors, *Proceedings of NCMA 2016*, number 321 in Österreichische Computer Gesellschaft, pages 211–226, 2016.

[18] LaSer. Independent LAnguage SERver. URL address: `http://laser.cs.smu.ca/independence/` Accessed in December of 2017.

[19] Vladimir I. Levenshtein. Certain properties of code systems. *Cybernetics and Control Theory, Soviet Physics Doklady*, 6:858–860, 1962. English translation of paper in *Dokl. Akad. Nauk. SSSR*, volume 140, pages 1274–1277, 1961.

[20] OpenFst Library. Google Research and NYU's Courant Institute. URL address: `http://www.openfst.org/` Accessed in December of 2017.

[21] Alexander A. Markov. Nonrecurrent coding. *Problems of Cypernetics*, 8:169–180, 1962. In Russian.

[22] Alexandru Mateescu and Arto Salomaa. Formal languages: an introduction and a synopsis. In Rozenberg and Salomaa [29], pages 1–39.

[23] Robert McCloskey. An $O(n^2)$ time algorithm for deciding whether a regular language is a code. *Journal of Computing and Information*, 2:79–89, 1996.

[24] Maurice Nivat. Elements de la théorie générale des codés. In E. Caianiello, editor, *Automata Theory*, pages 278–294. 1966.

[25] Filip Paluncic, Khaled Abdel-Ghaffar, and Hendrik Ferreira. Insertion/deletion detecting codes and the boundary problem. *IEEE Trans. Information Theory*, 59(9):5935–5943, 2013.

[26] PyPy. The official home of the pypy. URL address: `https://pypy.org/` Accessed in December of 2017.

[27] Python. The official home of the python programming language. URL address: `https://www.python.org/` Accessed in December of 2017.

[28] Darrell Raymond and Derick Wood. Grail: A C++ library for automata and expressions. *Journal of Symbolic Computation*, 17(4):341 – 350, 1994.

[29] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, Vol. I.* Springer-Verlag, Berlin, 1997.

[30] Jacques Sakarovitch. *Elements of Automata Theory.* Cambridge University Press, Berlin, 2009.

[31] August Albert Sardinas and George W. Patterson. A necessary and sufficient condition for the unique decomposition of coded messages. *IRE Intern. Conven. Rec.*, 8:104–108, 1953.

[32] Marcel Schützenberger. Une théorie algébrique du codage. In *Séminaire Dubreil-Pisot*, page Expose No. 15. 1955-56.

[33] H. J. Shyr. *Free Monoids and Languages.* Hon Min Book Company, Taichung, second edition, 1991.

[34] H. J. Shyr and Gabriel Thierrin. Codes and binary relations. In Marie Paule Malliavin, editor, *Séminaire d'Algèbre Paul Dubreil, Paris 1975–1976 (29ème Année)*, volume 586 of *Lecture Notes in Mathematics*, pages 180–188, 1977.

[35] Western University. Grail+. URL address: `http://www.csit.upei.ca/~ccampeanu/Grail/` Accessed in February, 2016.

[36] Vaucanson. The vaucanson project. URL address: `http://vaucanson-project.org/` Accessed in December of 2017.

[37] Margus Veanes. Applications of symbolic finite automata. In S. Konstantinidis, editor, *Proceedings of CIAA 2013*, volume 7982 of *Lecture Notes in Computer Science*, pages 16–23, 2013.

[38] Derick Wood. *Theory of Computation.* Harper & Row, New York, 1987.

[39] Sheng Yu. Regular languages. In Rozenberg and Salomaa [29], pages 41–110.

[40] Shyr Shen Yu. *Languages and codes.* Tsang Hai Book Publishing, Taichung, 2005.