Symbolic Manipulation of Code Properties

Stavros Konstantinidis[1], Casey Meijer[1], Nelma Moreira[2], Rogério Reis[2]

[1] Department of Mathematics and Computing Science, Saint Mary's University, Halifax, Nova Scotia, Canada, s.konstantinidis@smu.ca, dylanyoungmeijer@gmail.com

[2] CMUP & DCC, Faculdade de Ciências da Universidade do Porto, Rua do Campo Alegre, 4169007 Porto Portugal {nam,rvr}@dcc.fc.up.pt

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 improveded algorithms on these objects.

Keywords. algorithms, automata, codes, FAdo, implementation, language properties, LaSer, maximal, regular languages, transducers, program generation

1 Introduction

Several programming platforms are nowadays available, providing methods to transform and manipulate various formal language objects: Grail/Grail+ [26,33], Vaucanson 2 [6,34], FAdo [2,10], OpenFST [19], JFLAP [19]. 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 command shell. Similarly, FAdo provides a set of methods manipulating such objects on a Python shell [25].

Software environments for symbolic manipulation of formal languages are widely recognized as important tools for theoretical and practical 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 states of a minimal deterministic finite automata (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 add.

In this work, we extend the capabilities of FAdo and LaSer [9,17] 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.

1. Implementation of transducer objects and state of the art transducer methods. FAdo is a regularly maintained and user friendly Python package that was lacking transducer objects. Now available are general transducers as well as transducers in standard and normal forms. Some important methods that have been implemented are various product constructions between transducers and between transducers and automata, as well as a transducer functionality test.

2. 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. In simple mathematical terms, a code property is a class of languages that is closed for maximal languages. In addition to some fixed known code properties (such as prefix code, suffix code, hypercode), these methods can be used to construct new code properties, including various error-detecting properties, which are specified either via a trajectory regular expression [7] or via a transducer description [9]. Moreover, our methods can be used to combine any defined properties and produce a new property that is the conjunction of the defined properties.

3. Enhancement and implementation of state of the art 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. In particular, the satisfaction of the classic property of unique decipherability, or decodability, is implemented for any given NFA based on the algorithm in [12] as well as the satisfaction of various error-detecting properties [16].

4. 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.

5. Generation of executable Python code based on the requested question about a given code property. This is mostly of use in the online LaSer [17], which receives client requests and attempts to compute answers. However, as the algorithm required to compute an answer can take a long time, LaSer provides the option to compute and return a self-contained executable Python program that can be executed at the client's machine and return the required answer.

6. All the above classes and methods are open source (GPL): available for anyone to copy, use and modify to suit their own application.

Our work is founded on independence theory [15,32] as well as the theory of rational relations and transducers [4,28]. We present our algorithmic enhancements in a detailed mathematical manner with *two aims in mind:* first to establish the correctness of the enhanced algorithms and, second to allow interested readers to obtain a deeper understanding of these algorithms, which could potentially lead to further developments.

The paper is organized as follows.

Section 2 contains some basic terminology and background about various formal language concepts as well as a few examples of manipulating FAdo automata in Python.

Section 3 describes our implementation of transducer object classes and a few basic methods involving product constructions and rational operations.

Section 4 describes the 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 basic methods for their manipulation. Moreover, a mathematical approach to defining syntactic simulations of infinite sets of properties is presented, explaining that a linear simulation of all error-detecting properties exists, but no complete simulation of these properties 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 witnesses when the answer to the satisfaction/maximality question is negative.

Section 7 describes the implementation of the unique decodability (or decipherability) property and its satisfaction and maximality algorithms. This property is presented separately, as it is a classic one that cannot be defined within the methods of transducer properties.

Section 8 describes the new version of LaSer [17] including the capability to generate executable programming code in response to a client's request for the satisfaction or maximality of a given code property.

Section 9 contains a few concluding remarks including directions for future research.

2 Terminology and Background

Sets, alphabets, words, languages. We write $\mathbb{N}, \mathbb{N}_0, \mathbb{Z}, \mathbb{R}$ for the sets of natural numbers (not including 0), non-negative integers, integers, and real numbers, 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. If $q \in \mathbb{N}$, then Σ_q denotes the alphabet $\{0, 1, \ldots, q-1\}$. 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. In the rest of this paragraph, we use the following arbitrary object names: i, j for nonnegative integers, K, L for languages and u, v, w, x, y for 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, 21, 27, 36]. For example, uv, w^i, KL, L^i, L^*, L^+ represent respectively, the concatenation of u and v, the word consisting of i copies of w, the concatenation of K and L, the language consisting of all words obtained by concatenating any i L-words, the Kleene star of L, and $L^+ = L^* \setminus \varepsilon$. 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* $\mathcal{P}$. Let $\aleph_0$ denote the cardinality of $\mathbb{N}$. 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 the property exactly when all nonempty subsets of L with less than n elements satisfy the property. 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 $w \notin L$. From [15] we have that every L satisfying $\mathcal{P}$ is included in a maximal $\mathcal{P}$ code. To our knowledge, all known code related properties in the literature [5, 7, 9, 11, 15, 24, 31, 38] are code properties as defined above. 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 $n = 3$. Indeed, let $\mathcal{P}'$ be the set of all singleton languages union with the set of all languages $\{u, v\}$ such that u is not a proper prefix of v and vice versa. Then, every element in $\mathcal{P}'$ is a prefix code. Moreover any L is a prefix code if and only if any nonempty subset L' of L with less than three elements is in $\mathcal{P}'$. 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 [32].

Automata and regular languages [28, 37]. 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*. Let (p, x, q) be a transition of $\mathbf{a}$. Then x is called 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 nonnegative integer ℓ, such that each triple (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 ε-NFA $\mathbf{a}$ is called *trim*, if every state appears in some accepting path of $\mathbf{a}$. The automaton $\mathbf{a}$ is called an *NFA*, if no transition label is empty, that is, $T \subseteq Q \times \Sigma \times Q$. A deterministic finite automaton, or *DFA* for short, is a special type of NFA where 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 $|Q| + |T|$.

Transducers and (word) relations [4, 28, 37]. 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 (finite) *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* here is (x/y), with x being the input and y being the output label. 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 $(x_1 \cdots x_\ell, y_1 \cdots y_\ell)$ consisting of the input and output labels in the path. The *relation realized* by the transducer $\mathbf{t}$, denoted as $\mathrm{R}(\mathbf{t})$, is the set of labels in 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 $(x, y) \in \mathrm{R}(\mathbf{t})$. The *domain* of $\mathbf{t}$ is the set of all words w such that $\mathbf{t}(w) \neq \emptyset$. The *inverse* of a transducer $\mathbf{t}$, denoted as $\mathbf{t}^{-1}$, is the transducer that results from $\mathbf{t}$ by simply switching the input and output alphabets of $\mathbf{t}$ and also switching the input and output parts of the labels in the transitions of $\mathbf{t}$. It follows that $\mathbf{t}^{-1}$ realizes the inverse of the relation realized by $\mathbf{t}$. 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. 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]. The modules `fa` for automata, `fl` for finite languages, and `fio` for input/output of formal language objects, can be imported in a standard Python manner as follows.

```
import FAdo.fl as fl
import FAdo.fio as fio
from FAdo.fa import *    # import all fa methods for readability
```

The FAdo object classes `FL`, `DFA` and `NFA` manipulate finite languages, DFAs and ε-NFAs, respectively.

Example 1. The following code builds a finite language object `L` from a list of strings, and then builds an NFA object `a` accepting the language $\{a, ab, aab\}$.

```
lst = ['a', 'ab', 'aab']
L = fl.FL(lst)
a = L.toNFA()
```

The second line uses the class name `FL` to create the finite language `L` from the given list of strings. The last line returns an NFA accepting the language `L`. □

The following code reads an automaton, or transducer, `a` from a file, and then writes it into a file.

```
a = fio.readOneFromFile(filename1)
 ...
fio.saveToFile(filename2, a)
```

These methods assume that automata and transducers are written into those files in FAdo format—see examples below and further ones in [10, 17]. We can also read an automaton or transducer from a string `s` using the function `readOneFromString(s)`.

Example 2. The following code defines a string that contains an automaton accepting a^*b and then uses that string to define an `NFA` object.

```
st = '@NFA 1 * 0\n0 a 0\n0 b 1\n'
a = fio.readOneFromString(st)
```

As usual, the pattern `\n` denotes the *end of line character*, so the string `st` consists of 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`. The string has to end with a `\n`. □

Next we list a few useful methods for `NFA` objects. We assume that `a` and `b` are automata, and `w` is a string (a word).

`a.evalWordP(w)`: returns `True` or `False`, depending on whether the automaton `a` accepts `w`.

`a.emptyP()`: returns `True` or `False`, depending on whether the language accepted by `a` is empty.

`a & b`: returns an NFA accepting $\mathrm{L}(\mathtt{a}) \cap \mathrm{L}(\mathtt{b})$. Both `a` and `b` must be NFAs with no ε-transitions.

`a.elimEpsilon()`: alters `a` to an equivalent NFA with no ε-transitions.

`a.epsilonP()`: returns `True` or `False`, depending on whether `a` has any ε-transitions.

`a.makePNG(fileName='xyz')`: creates a file `xyz.jpg` containing a picture of the automaton (or transducer) `a`.

`a.enumNFA(n)`: returns the set of words of length up to `n` that are accepted by the automaton `a`.

Example 3. The following example shows a naive 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 the automaton accepting {`w`} accepts something. □

3 Transducer Object Classes and Methods

In this section we discuss some aspects of the implementation of transducer objects and then continue with several important methods, which we divide into two subsections: 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`. We can import all transducer methods as follows.

```
from FAdo.transducers import *
```

Transducer objects and basic methods

From a mathematical point of view, a *Python dictionary* is an onto function `delta` : $D \to R$ between two finite sets of values D and R. One writes `delta[x]` for the image of `delta` on `x`. One can define completely the dictionary using an expression of the form

```
delta = {d1:r1, d2:r2, ... , dN:rN}
```

which defines `delta[di] = ri`, for all $i = 1, \ldots, \mathrm{N}$. In FAdo, the class `GFT`, for General Form Transducer, is 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 $\mathbf{t}$. Specifically, `States` is a list of unique state names, meaning that each state name has an index which is the position of the state name in the list, with 0 being the first index value. The variables `Sigma, Output, Initial` and `Final` are sets, where the latter two are sets of state indexes. For efficiency reasons, the set of transitions T is implemented as a dictionary

`delta`: $\{0, \ldots, n-1\} \to (\texttt{Sigma} \to 2^{\texttt{Output} \times \{0,\ldots,n-1\}})$,

where n is the number of states. Thus, for any $p \in \{0, \ldots, n-1\}$, `delta[`p`]` is a dictionary, and for any input label x, `delta[`p`][`x`]` is a set of pairs (y, q) corresponding to all transitions $\{(p, x/y, q) \in T \mid y \in \texttt{Output}, q \text{ is a state index}\}$.

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

```
s = t.toSFT()
```

which assigns to `s` a new SFT object 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 [28].

Example 4. The following code defines a string `s` that contains a transducer description and then constructs an SFT transducer from that string. The transducer, on input x, returns the set of all proper suffixes of x—see also Fig 1. It has an initial state 0 and a final state 1, and deletes at least one of the input symbols so that what gets outputted at state 1 is a proper suffix of the input word.

```
s = '@Transducer 1 * 0\n'\
    '0 a @epsilon 0\n'\
    '0 b @epsilon 0\n'\
    '0 a @epsilon 1\n'\
    '0 b @epsilon 1\n'\
    '1 a a 1\n'\
    '1 b b 1\n'
t = fio.readOneFromString(s)
```

□

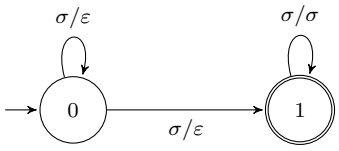

Figure 1: On input x the transducer shown in the figure outputs any proper suffix of x. Note: In this and the following transducer figures, the input and output alphabets are equal. An arrow with label σ/σ represents a set of transitions with labels σ/σ, for all alphabet symbols σ; and similarly for 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'.

Recall, for a transducer `t` and word w, $\mathbf{t}(w)$ is the set of possible outputs of `t` on input w. Note that this set could be empty, finite, or even infinite. In any case, it is always a regular language. The FAdo method `t.runOnWord(w)` assumes that `t` is an SFT object and returns an automaton accepting the language t(w).

Example 5. The following code is a continuation of Example 4. It prints the set of all proper suffixes of the word `ababb`.

```
a = t.runOnWord('ababb')
n = len('ababb')
print a.enumNFA(n)
```

□

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

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

`t.evalWordP((w,w'))`: returns `True` or `False`, depending whether the pair `(w,w')` belongs to the relation realized by `t`.

`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, 37]

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 the automaton accepting the language

$$\bigcup_{w \in \mathrm{L}(\mathbf{a})} \mathbf{t}(w).$$

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

To make the presentation a little more concrete for interested readers, we comment on one of the above methods, `t.runOnNFA(a)`. The construction first considers whether `t` has any ε-input transitions, that is transitions with labels ε/y. If yes, then a copy `b` of `a` is made with ε-loops added, that is, transitions (p, ε, p) for all states p in `a`. Then, if `a` has any ε-transitions then a copy `s` of `t` is made with ε-loops added, that is, transitions $(q, \varepsilon/\varepsilon, q)$ for all states q in `t`. Then, the actual product construction is carried out: if (p, x, p') and $(q, x/y, q')$ are transitions of `b` and `s`, respectively, then $((p, q), y, (p', q'))$ is a transition of the resulting automaton.

Example 6. Continuing Examples 2 and 4, the following code constructs an NFA object `b` accepting every word that is a proper suffix of some word in a^*b. It then enters a loop that prints whether a given string w is a suffix of some word in a^*b.

```
b = t.runOnNFA(a)
while True:
    w = raw_input()
    if 'a' not in w and 'b' not in w: break
    print b.evalWordP(w)
```

□

Rational operations [4]

A relation ρ is a *rational relation*, if it is equal to $\emptyset$, or $\{(x, y)\}$ for some words x and y, or can be obtained from other ones by using a finite number of times any of the three (rational) operators: union, concatenation, Kleene star. A classic result on transducers says that a relation is rational if and only if it can be realized by a transducer. The following methods are now available in FAdo, where we assume that `s` and `t` are SFT transducers.

`t.union(s)`: returns a transducer realizing the union of the relations realized by `s` and `t`.

`t.concat(s)`: returns a transducer realizing the concatenation of the relations realized by `s` and `t`.

`t.star(flag=False)`: returns a transducer realizing the Kleene star of the relation realized by `t`, assuming the argument is missing or is `False`. Else it returns a transducer realizing $(\mathrm{R}(\mathtt{t}))^+$.

The implementation of the above methods mimics the implementation of the corresponding methods on automata.

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 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—see [1].

Given a 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

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

2. 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 $\mathbf{t}$.

3. 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

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

5. Starting from the initial states, visit all transitions in breadth-first search mode such that, if (p,p') has the 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.

6. 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.

7. 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.

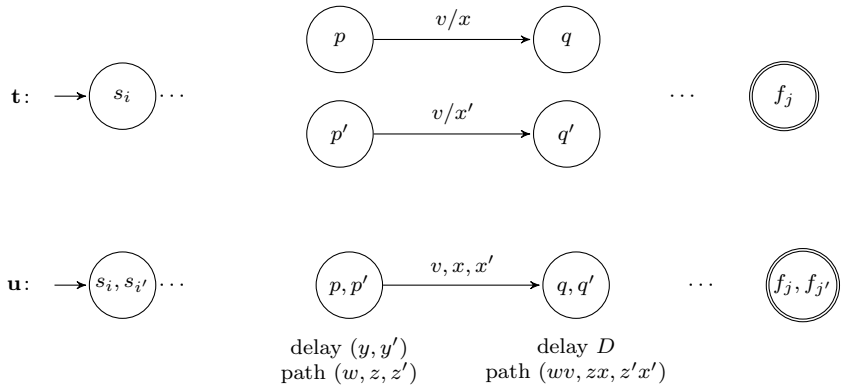

Figure 2: The figure shows the structure of the (revised) square machine $\mathbf{u}$ corresponding to the given transducer $\mathbf{t}$. The delay and path values for the states of $\mathbf{u}$ are explained in Definition 7

Definition 7. Let (q,q') be a state of the new 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.
  1. If $C=(y,y')\neq$ ZERO and $y'x'$ is of the form yxu then $D=(\varepsilon,u)$.
  2. If $C=(y,y')\neq$ ZERO and yx is of the form $y'x'u$ then $D=(u,\varepsilon)$.
  3. If $C=(y,y')\neq$ ZERO and $y'x'=yx$ then $D=(\varepsilon,\varepsilon)$
  4. 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 $\mathbf{u}$.

Remark 8. 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 $\mathbf{u}$ whose label is (w,z,z'). Moreover, by the definition of $\mathbf{u}$, the transducer $\mathbf{t}$ 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

1. Define function `completePath`(q,q') that follows a shortest path from (q,q') to a final state of $\mathbf{u}$ and returns a suffix triple (see Definition 7).
2. Construct the revised square machine $\mathbf{u}$, 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).
3. Assign to each initial state the delay-path value $((\varepsilon,\varepsilon),(\varepsilon,\varepsilon,\varepsilon))$.
4. 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 7, 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.
5. At this point `return (None,None,None)`, as the breadth-first process has been completed.

Proposition 9. *Algorithm* `nonFunctionalW` *takes as input a standard form transducer* $\mathbf{t}$ *and returns either a witness of* $\mathbf{t}$*'s non-functionality, or the triple* `(None,None,None)` *if* $\mathbf{t}$ *is functional.*

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

Lemma 10. *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 7. 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 7 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 Prop. 9) 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 $\mathbf{t}$'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 $\mathbf{t}$'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 $\mathbf{t}$'s non-functionality

$$(\alpha w_{qq'}, \beta z_{qq'}, \beta' z'_{qq'}), \quad (\alpha_1 w_{qq'}, \beta_1 z_{qq'}, \beta'_1 z'_{qq'}).$$

For statement S1, by Remark 8, 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 $\alpha = wv$, $\beta = zx$, $\beta' = z'x'$. Now using the above lemma we get, for some h

$\beta z_{qq'} = zxz_{qq'} = hyxz_{qq'} = hf\sigma g z_{qq'}$ 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 8, 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 $\alpha = wv$, $\beta = zx$, $\beta' = z'x'$. Using the above lemma we get, for some h

$\beta = zx = hyx$ 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 $\beta_1 z_{qq'} = \beta_1 z'_{qq'}$. Using the above lemma we get, for some h

$\beta = hs$, $\beta' = hs'$, $\beta_1 = h_1 s_1$, $\beta'_1 = h_1 s'_1$.

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. □

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 of course one can only implement countably many properties. So we are interested in systematic methods that allow one to formally describe code properties. Three such 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. These formal methods appear to be able to describe most properties of practical interest. 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. The implementation of implicational conditions is an interesting topic for future research.

Next we review quickly the formal methods of regular trajectories and transducers, and then discuss our implementation of these formal methods.

Regular trajectory properties [7]. In this formal method a regular expression $\bar{e}$ over $\{0,1\}$ describes the code property $\mathcal{P}_{\bar{e}}$ as follows. The 0s in $\bar{e}$ indicate positions of alphabet symbols that make up a word v, say, and the 1s in $\bar{e}$ indicate positions of arbitrary symbols that, together with the 0s, make up a word u, say. A language L satisfies $\mathcal{P}_{\bar{e}}$ if there are no two different words in $u, v \in L$ such that u has the structure indicated by $\bar{e}$ and v has a structure obtained by deleting the 1s from $\bar{e}$. For example, the *infix code* property is defined 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

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

With this formal method we can define the *suffix code* property: L is a suffix code if no L-word is a proper suffix of an L-word. The transducer defined in Example 4 is input-altering and describes the suffix code property over the alphabet {`a, b`}. 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 always be 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 $\mathcal{P}^{\mathrm{ed}}_{\mathbf{t}}$ consisting of all languages L over the input alphabet of $\mathbf{t}$ such that

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

The term error-detecting ***for*** $\mathbf{t}$ is used in the sense that L is meant to consist of all valid messages one can transmit via $\mathbf{t}$, and $\mathbf{t}$ cannot turn a valid message into a different valid message. We note that, for every input-altering transducer $\mathbf{t}$, one can make in linear time a channel transducer $\mathbf{t}'$ such that $\mathcal{P}^{\mathrm{al}}_{\mathbf{t}} = \mathcal{P}^{\mathrm{ed}}_{\mathbf{t}'}$ [9]. Thus, every input-altering transducer property is an error-detecting property.

Example 11. 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 3. 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\n'\
       '0 b b 0\n'\
       '0 b a 1\n'\
       '0 a b 1\n'\
       '1 a a 1\n'\
       '1 b b 1\n'
t1 = fio.readOneFromString(s1)
```

□

We note that 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 3. Codes for such errors are actively under investigation—see [24], for instance.

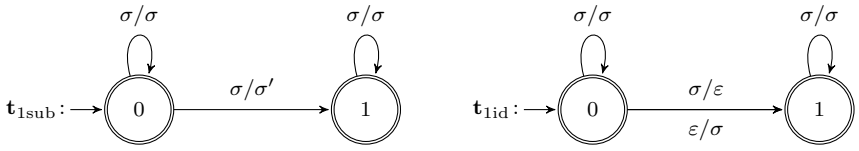

Figure 3: On input x the transducer $\mathbf{t}_{1\text{sub}}$ outputs either x, or any word that results by substituting exactly one symbol in x. On input x the transducer $\mathbf{t}_{1\text{id}}$ outputs either x, or any word that results by deleting, or inserting, exactly one symbol in x. Note: The use of labels on arrows is explained in Fig. 1.

Error-correcting properties via input-preserving transducers. An input-preserving (or channel) transducer is used to describe the *error-correcting for* $\mathbf{t}$ property $\mathcal{P}^{ec}_{\mathbf{t}}$ consisting of all languages L over the input alphabet of $\mathbf{t}$ such that

$$\mathbf{t}(v) \cap \mathbf{t}(w) = \emptyset, \quad \text{for all distinct words } v, w \in L. \tag{3}$$

The term error-correcting ***for*** $\mathbf{t}$ is used in the sense that any message w' received from an L-word via $\mathbf{t}$ can result from only one such L-word, so if $w' \notin L$ then w' can in principle be corrected to exactly one L-word.

Remark 12. It can be shown that a language is error-correcting for $\mathbf{t}$ if and only if it is error-detecting for any transducer realizing the relation $\mathrm{R}(\mathbf{t}^{-1}) \circ \mathrm{R}(\mathbf{t})$.

5.1 Implementation in FAdo.

We present now our implementation of the previously mentioned code properties. We have defined the Python classes

`TrajProp, IATProp, ErrDetectProp, ErrCorrectProp`

corresponding to the four types of properties discussed above. These four property types are described, respectively, by regular trajectory expressions, input-altering transducers, input-preserving

transducers, and input-preserving transducers. In all four cases, given a transducer object, an object of the class is created. An object `p` of the class `IATProp`, say, is defined via some transducer `t` and represents a particular code property, that is, the class of languages satisfying Eq. (1).

The class `ErrDetectProp` is a superclass of the others. These classes and all related methods and functions are in the module `codes.py` and can be imported as follows.

```
import FAdo.codes as codes
```

Although each of the above four classes requires a transducer to create an object of the class, 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 in use with specific arguments from previous examples.

Example 13. Consider again Examples 4 and 11 in which the strings `s` and `s1` are defined containing, respectively, the proper suffixes transducer and the transducer permitting up to 1 substitution error. The following object definitions are possible with the FAdo package

```
icp = codes.buildTrajPropS('1*0*1*', {'a', 'b'})
scp = codes.buildIATPropS(s)
s1dp = codes.buildErrorDetectPropS(s1)
s1cp = codes.buildErrorCorrectPropS(s1)
pcp = codes.buildPrefixProperty({'a', 'b'})
icp2 = codes.buildInfixProperty({'a', 'b'})
```

In the first statement, `icp` represents the infix code property over the alphabet {`a, b`} and is defined via the trajectory expression `1*0*1*`. In the next three statements, `scp, s1dp, s1cp` represent, respectively, the suffix code property, the 1-substitution error-detecting property and the 1-substitution error-correcting property. The last two statements are explained below—`pcp` and `icp2` represent the prefix code and infix code properties, respectively. □

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

`buildPrefixProperty(Sigma)`: returns an object of the class `PrefixProp` that represents all prefix codes over the alphabet `Sigma`.

`buildSuffixProperty(Sigma)`: returns an object of the class `SuffixProp` that represents all suffix codes over the alphabet `Sigma`.

`buildInfixProperty(Sigma)`: returns an object of the class `InfixProp` that represents all infix codes over the alphabet `Sigma`. Note that an infix code is both prefix and suffix and this fact is reflected in the class definition, by considering `InfixProp` a Python subclass of both `PrefixProp` and `SuffixProp`.

`buildOutfixProperty(Sigma)`: returns an object of the class `OutfixProp` that represents all outfix codes over the alphabet `Sigma`. A language L is an *outfix code* if deleting an infix of an L-word cannot result into another L-word. Note that an outfix code is both prefix and suffix, and, as above, `OutfixProp` is a subclass of the correspondent Python classes.

`buildHypercodeProperty(Sigma)`: returns an object of the class `HypercodeProp` that represents all hypercodes over the alphabet `Sigma`. A language L is a *hypercode* if deleting any symbols of an L-word cannot result into another L-word. Note that a hypercode is both infix and outfix and as above, `HypercodeProp` is a subclass of the correspondent Python classes.

Each of the above methods creates internally a transducer whose input and output alphabets are equal to the given alphabet `Sigma`, and then passes the transducer to the constructor of the respective class.

Combining code properties. In practice it is desirable to be able to talk about languages that satisfy more than one code property. For example, most of the 1-substitution error-detecting codes used in practice are infix codes (in fact *block codes*, that is, those whose words are of the same length). 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 14. Using the properties `icp, s1dp` created above in Example 13, 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—see Section 6. □

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

$$\mathcal{P}^{ed} = \{\mathcal{P}^{ed}_{\mathbf{t}} \mid \mathbf{t} \text{ is an input-preserving transducer}\}. \tag{4}$$

In fact, for `ErrDetectProp` objects, we have implemented the methods '&' and '$\leq$' in a way that the triple $(\mathtt{ErrDetectProp}, \&, \leq)$ constitutes a syntactic hierarchy (see next section) which can be used to simulate the properties in (4). 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. (4) 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 13.

The mathematical definitions and results about 'syntactic hierarchy', 'syntactic simulation' and 'complete syntactic simulation' are explained in the next section.

5.2 Mathematical Aspects of Code Hierarchy Implementation

Here we give a mathematical definition of what it means to simulate a set of code properties $\mathcal{Q} = \{\mathcal{Q}_j \mid j \in J\}$ via a syntactic hierarchy $(G, \&, \leq)$—see definition below—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 $x, y \in \langle G \rangle$.

The first result is that there is an efficient simulation of the set $\mathcal{P}^{ed}$ in Eq. (4)—see Theorem 18. The second result is 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 $x, y \in \langle G \rangle$—see Theorem 19.

Definition 15. A syntactic hierarchy is a triple $(G, \&, \leq)$ such that G is a nonempty set and

1. $(\langle G\rangle, \&)$ is the commutative semigroup generated by G with computable operation '&'.
2. $(\langle G\rangle, \leq)$ is a decidable partial order (reflexive, transitive, antisymmetric).
3. For all $x, y \in \langle G\rangle$, $x \leq y$ implies $x\&y = x$.
4. 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 16. *The following statements hold true, for all $x, y, z \in \langle G\rangle$,*

1. *$x \leq x$ and $x\&x = x$*
2. *$x \leq y$ if and only if $x = y\&z$ for some $z \in \langle G\rangle$.*
3. *$x = x\&y$ if and only if $x \leq y$.*
4. *If $x \leq y$ and $x \leq z$ then $x \leq y\&z$.*
5. *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.*
6. *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 the previous definition and uses standard logical arguments. We present only proofs for the second and fourth statements.

The 'if' part of the second statement follows from the fourth statement of the above definition, and the 'only if' part follows from the third statement of the above 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 17. Let $\mathcal{Q} = \{\mathcal{Q}_j \mid j \in J\}$ be a set of properties, for some index set J. A (syntactic) simulation of $\mathcal{Q}$ is a quintuple $(G, \&, \leq, [\,], \varphi)$ such that $(G, \&, \leq)$ is a syntactic hierarchy and

1. $[\,]$ is a surjective mapping of $\langle G\rangle$ onto $\mathcal{Q}$;
2. for all $x, y \in \langle G\rangle$, $x \leq y$ implies $[x] \subseteq [y]$;
3. for all $x, y \in \langle G\rangle$, $[x\&y] = [x] \cap [y]$;
4. φ is a computable function of J into $\langle G\rangle$ such that $[\varphi(j)] = \mathcal{Q}_j$.

The simulation is called *complete* if, for all x, y

$$[x] \subseteq [y] \quad \text{implies} \quad x \leq y.$$

The simulation is called linear if J has a size function $|\cdot|$ and $\langle G\rangle$ has a size function $\|\cdot\|$ such that $\|\varphi(j)\| = O(|j|)$, for all $j \in J$, and for all x, y

$$\|x\&y\| \;=\; O(\|x\| + \|y\|).$$

By a size function on a set X, we mean any function f of X into $\mathbb{N}_0$.

Theorem 18. *There is a linear simulation of the set*

$$\{\mathcal{P}^{\mathrm{ed}}_{\mathbf{t}} \mid \mathbf{t} \textit{ is an input-preserving transducer}\}.$$

Proof. Let $\mathbf{T}$ be the set consisting of all finite sets of transducers. Let $T_1, T_2 \in \mathbf{T}$. We define

$G = \{\{\mathbf{t}\} \mid \mathbf{t}$ is an input-preserving transducer$\}$.

$T_1 \& T_2 = T_1 \cup T_2$.

$T_1 \leq T_2$, 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 15 are satisfied. Thus $(G, \&, \leq)$ is a syntactic hierarchy.

Next we define the size function. For a finite nonempty set $T = \{\mathbf{t}_1, \ldots, \mathbf{t}_n\}$ of transducers, we denote with $\vee T$ the transducer $\mathbf{t}_1 \vee \cdots \vee \mathbf{t}_n$ of size $O(\sum_1^n |\mathbf{t}_i|)$ realizing $\mathrm{R}(\mathbf{t}_1) \cup \cdots \cup \mathrm{R}(\mathbf{t}_n)$. Then, define $\|T\| = |\vee T|$. One verifies that $\|T_1 \& T_2\| = O(\|T_1\| + \|T_2\|)$ as required.

Next we use the syntactic hierarchy $(G, \&, \leq)$ to define the required simulation. First, define $\varphi(\mathbf{t}) = \{\mathbf{t}\}$, for any input-preserving transducer $\mathbf{t}$. Then, define

$$[T] = \mathcal{P}^{ed}_{\vee T}, \quad \text{which equals} \quad \bigcap_{\mathbf{t} \in T} \mathcal{P}^{ed}_{\mathbf{t}}.$$

One verifies that the requirements of Definition 17 are satisfied. □

The next result shows that there can be no complete simulation of the set of error-detecting properties. This follows from the undecidability of the Post Correspondence problem using methods from establishing the undecidability of basic transducer related problems [4].

Theorem 19. *There is no complete simulation of the set of properties*

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

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

Lemma 20. *For any input-preserving transducers* $\mathbf{t}, \mathbf{s}$ *we have*

$$\mathcal{P}^{\mathrm{ed}}_{\mathbf{t}} \subseteq \mathcal{P}^{\mathrm{ed}}_{\mathbf{s}} \text{ if and only if } \mathrm{R}(\mathbf{s} \vee \mathbf{s}^{-1}) \subseteq \mathrm{R}(\mathbf{t} \vee \mathbf{t}^{-1}).$$

Proof. First note that Eq. (2) is equivalent to

$$(w, v) \in \mathrm{R}(\mathbf{t}) \text{ implies } w = v, \quad \text{for all } v, w \in L.$$

This implies that, for all input-preserving transducers $\mathbf{t}, \mathbf{s}$, we have $\mathcal{P}^{ed}_{\mathbf{s}} = \mathcal{P}^{ed}_{\mathbf{s}^{-1}}$ and

$$\mathrm{R}(\mathbf{s}) \subseteq \mathrm{R}(\mathbf{t}) \quad \text{implies} \quad \mathcal{P}^{ed}_{\mathbf{t}} \subseteq \mathcal{P}^{ed}_{\mathbf{s}}, \tag{5}$$

$$\mathcal{P}^{ed}_{\mathbf{s}} \quad = \quad \mathcal{P}^{ed}_{\mathbf{s} \vee \mathbf{s}^{-1}}. \tag{6}$$

The statement of the lemma follows from the above observations using standard set theoretic arguments. □

Lemma 21. *The following problem is undecidable.*

Input: input-preserving transducers $\mathbf{t}, \mathbf{s}$

Return: whether $\mathrm{R}(\mathbf{s} \vee \mathbf{s}^{-1}) \subseteq \mathrm{R}(\mathbf{t} \vee \mathbf{t}^{-1})$

Proof. We reduce the Post Correspondence Problem (PCP) to the given problem. Consider any instance $((u_i)_1^p, (v_i)_1^p)$ of PCP which is a pair of sequences of p nonempty words over some alphabet B, for some positive integer p. As mentioned before, we use tools that have been used in showing the undecidability of basic transducer problems [4]. In particular, 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

$$\begin{aligned} C &= \{ab, ab^2, \ldots, ab^p\} && (7)\\ \mathrm{diag}(L) &= \{(x, x) \mid x \in L\}, \text{ for any language } L && (8)\\ D &= \mathrm{diag}(C^+) \cup \mathrm{diag}(aaB^+) && (9)\\ X &= (\varepsilon, aa)U^+ \cup D && (10)\\ Y &= ((C^+ \times aaB^+) - (\varepsilon, aa)V^+) \cup D && (11)\\ \mathbf{t} &= \text{ any transducer realizing X} && (12)\\ \mathbf{s} &= \text{ any transducer realizing Y} && (13) \end{aligned}$$

We use rational relation theory to show the following claims, which establish the required reduction from PCP to the given problem.

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: First note that, as C^+ and aaB^+ are regular languages, we have that D is a rational relation. In [4], the author 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 it is easy to confirm that $X^{-1} \subseteq Y^{-1}$ if and only if $X \subseteq Y$ (in fact for any relations X and Y). Also the fact that $C^+ \cap aaB^+ = \emptyset$ 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 19.) For the sake of contradiction, assume there is a complete simulation $(G, \&, \leq, [\,], \varphi)$ of the given set of properties. Then, for all $x, y \in \langle G \rangle$, we have

$$[x] \subseteq [y] \text{ implies } x \leq y.$$

We derive a contradiction by showing that the problem in Lemma 21 is decidable 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 set 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).

It is assumed that the code property can be implemented as `p` via a transducer $\mathbf{t}$ (whether input-altering or input-preserving) and, 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 $\mathbf{b}$. In fact one can give a language via a regular expression, in which case it is converted to an automaton. Next we present the implementation of methods for the satisfaction and maximality problems. We discuss briefly the construction problem in the last section of the paper.

Methods `p.satisfiesP(a)`

The satisfaction problem is decidable in time $O(|\mathbf{t}||\mathbf{a}|^2)$, if the property `p` is of the input-altering transducer type. This follows from Eq. (1) and can be implemented as follows

```
c = t.runOnNFA(a)
return (a & c).emptyP()
```

For the case of `p` being the error-detecting property, the transducer $\mathbf{t}$ is input-preserving and Eq. (2) is decided via a transducer functionality test [16]. In FAdo this test can be implemented as follows, where the method `functionalP()` returns whether the transducer is functional.

```
s = t.inIntersection(a)
return s.outIntersection(a).functionalP()
```

For the case of `p` being the error-correcting property, again the given transducer is input-preserving and Eq. (3) is decided via a transducer functionality test [16]. In FAdo this test can be implemented as follows

```
s = t.inverse()
return s.outIntersection(a).functionalP()
```

Method `p.maximalP(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

$$\mathrm{L}(\mathbf{b}) \setminus (\mathrm{L}(\mathbf{a}) \cup \mathbf{t}(\mathbf{a}) \cup \mathbf{t}^{-1}(\mathbf{a})) = \emptyset. \tag{14}$$

The above emptiness test is implemented in the method `p.maximalP(a, b)`, which returns `True` or `False`, and uses standard NFA methods as well as the transducer methods `t.inverse()` and `t.runOnNFA(a)`. For the case of error-correcting properties, our implementation makes use of Remark 12.

Methods with witnesses: `p.notMaximalW(a, b)`

It can be shown that any word belonging to the set shown in Eq. (14) can be added to $\mathrm{L}(\mathbf{a})$ and the resulting language will still satisfy the property [9]. This word can serve as a witness of the non-maximality of $\mathrm{L}(\mathbf{a})$. If no such word exists, the method returns `None`.

Methods with witnesses: `p.notSatisfiesW(a)`

For input-altering transducer and error-detecting properties, the witness version of the method `p.satisfiesP(a)` returns either a pair of *different* words $u, v \in \mathrm{L}(\mathtt{a})$ violating the property, that is, $v \in \mathtt{t}(u)$ or $u \in \mathtt{t}(v)$, or they return the pair $(\mathtt{None}, \mathtt{None})$. In the former case, the pair (u, v) is called a *witness of the non-satisfaction of* `p` *by* the language $\mathrm{L}(\mathtt{a})$. For error-correcting properties `p`, a witness of non-satisfaction by $\mathrm{L}(\mathtt{a})$ is a triple of words (z, u, v) such that $u, v \in \mathrm{L}(\mathtt{a})$ and $u \neq v$ and $z \in \mathtt{t}(u) \cap \mathtt{t}(v)$. Next we discuss how to accomplish this by changing the implementations of `p.satisfiesP(a)` shown before.

Case 1: For input-altering transducer properties, we have the Python 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 $\mathrm{L}(\mathtt{a})$. This witness corresponds to the emptiness test in Eq. (1), as required.

Case 2: For error-detecting properties, 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 the method `nonFunctionalW()` returns a triple of words (u, v, w) then, by Proposition 9 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 $\mathrm{L}(\mathtt{a})$. 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 the non-functionality method returns a triple of `None`s then the constructed transducer is not functional. Then $\mathrm{L}(\mathtt{a})$ must satisfy the property, otherwise any different words $v, w \in \mathrm{L}(\mathtt{a})$ 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.

Case 3: For error-correcting properties, we use again the non-functionality witness method.

```
return t.inverse().outIntersection(a).nonFunctionalW()
```

For the correctness of this algorithm, one argues similarly as in the previous case.

The above discussion establishes the following consequence of Proposition 9 and the definitions of product constructions in Section 3.

Proposition 22. *The algorithms implemented in the three forms (input-altering transducer, error-detecting, error-correcting) of the method* `p.notSatisfiesW(a)` *correctly returns a witness of the non-satisfaction of the property* `p` *by the language* L(`a`).

Example 23. The following Python interaction shows that the language a^*b is a prefix and 1-error-detecting code. Recall from previous examples that the Python 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 in coding theory from the points of view of both information theory [29] as well as formal language theory [23,30]. 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 the two sequences are identical. In simpler terms, every word in L^* can be decomposed in exactly one way as a sequence of L-words.

In this section we describe our implementation of the satisfaction and maximality methods for the UD code property. We assume that `a` is an NFA object without ε-transitions. One can create the UD code property using the following syntax

```
p = codes.buildUDCodeProperty(a.Sigma)
```

The method `p.notSatisfiesW(a)`

The satisfaction problem for this property was discussed first in the well known paper [29], where the authors produce a necessary and sufficient condition for a finite language to be a UD code—we note that some feel that that condition does not clearly give an algorithm, as for instance the papers [18,20]. Over the years people have established faster algorithms and generalized the problem to the case where the language in question is regular. To our knowledge, the asymptotically most efficient algorithms for regular languages are given in [12,22], and they are both of quadratic time complexity. Our implementation follows the algorithm in [12], as that approach makes use of the transducer functionality algorithm. As before we enhance that algorithm to produce a *witness of non-satisfaction* which, given an NFA object `a`, it either returns the pair (`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).

We now describe the algorithm in [12] modified appropriately to return a witness of non-satisfaction. Again, the heart of the algorithm is the transducer functionality method. Let $\mathbf{a} = (Q, \Sigma, T, I, F)$ be the given NFA (with no ε-transitions).

1. If any of the initial states is final, then (as ε is in the language) return ($[\varepsilon], [\varepsilon, \varepsilon]$).
2. Construct the transducer $\mathbf{t} = (Q, \Sigma, \{0,1\}, T', I, I)$ in which the transition set T' is defined by the following process: If $(p, \sigma, q) \in T$ then $(p, \sigma/0, q) \in T'$ and, if in addition $q \in F$ then also $(p, \sigma/1, i) \in T'$, for every $i \in I$. Note that the domain of the transducer is exactly the language $\mathrm{L}(\mathbf{a})^*$.
3. Let `w, x, y = t.nonFunctionalW(a)`
4. If any of `w, x, y` is `None` then return (`None`, `None`)
5. At this point, we know that $w \in \mathrm{L}(\mathbf{a})$, x and y are different and each one is of the form

$$0^{r_1}1 \cdots 0^{r_n}1.$$

   Moreover, each of x and y corresponds to a decomposition of w in terms of $\mathrm{L}(\mathbf{a})$-words. More specifically, the binary word $0^{r_1}1 \cdots 0^{r_n}1$ encodes a sequence of words $(w_i)_1^n$ such that their concatenation is equal to w and each w_i is the infix of w that starts at position s_i and ends at position $s_i + r_i$, where $s_1 = 0$ and $r_i = |w_i| - 1$ and $s_{i+1} = s_i + r_i + 1$. For example, if $w = ababab$ and $x = 010001$, then the decomposition is $(ab, abab)$. The algorithm returns the two lists of words corresponding to x and y.

Example 24. 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 fundamental theorem of Schutzenberger [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.

```
t = infixTransducer(a.Sigma, True)
b = a.star()
return (~(t.runOnNFA(b))).emptyP()
```

The first statement above returns a transducer $\mathbf{t}$ that, on input w, outputs any infix of w. The second statement returns an NFA accepting $\mathrm{L}(\mathtt{a})^*$, and the last statement returns whether there is no word in the complement of all infixes of $\mathrm{L}(\mathbf{a})^*$.

8 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 client uploads a file containing an automaton in Grail format and a file containing either a trajectory automaton, or an input altering-transducer, and LaSer would respond with an answer to the witness version of the satisfaction problem for input-altering transducer properties. The new version, which we discuss here, is based on the FAdo set of automaton and transducer methods and allows clients 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 can be downloaded on the client's machine and executed on that machine returning thus the desired answer. This feature is useful as the execution of certain algorithms, even of polynomial time complexity—see the error-detection satisfaction problem–, can be quite time consuming for a server software like LaSer.

The user interface of LaSer is very simple—see Fig. 4. The user provides the file containing the

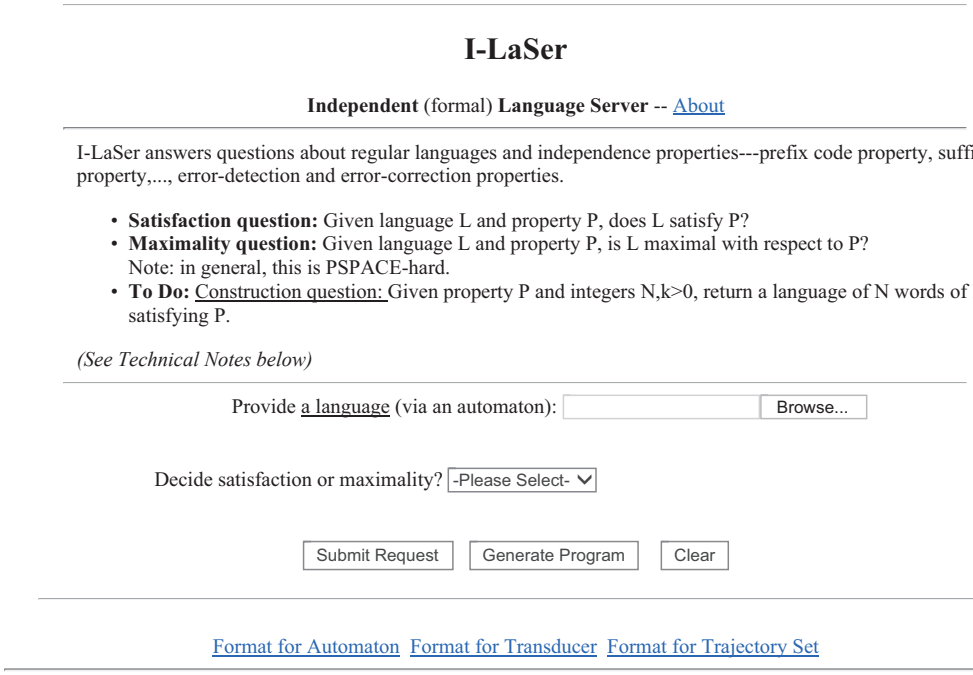

Figure 4: The main user interface of LaSer

automaton whose language will be tested for a question (satisfaction or maximality) about some property. When the decision question is chosen, then the user is asked to enter the type of property, which can be one of fixed, trajectory, input-altering transducer, error-detecting, error-correcting. Then, the user either clicks on the button "Submit Request" or on the button "Generate Program".

Next we present some parts of the program generation module. The program to be generated will contain code to answer one of the satisfaction or maximality questions for any of the properties regular trajectory (TRAJECT), input-altering transducer (INALT), error-detecting (ERRDET), error-correcting (ERRCORR), or any of the fixed properties. First a Python dictionary is prepared to enable easy reference to the property type to be processed, and another dictionary for the question to be answered, as follows

```
buildName = {"CODE": ("buildUDCodeProperty", ["a.Sigma"], 1),
             "ERRCORR": ("buildErrorCorrectPropS", ["t"], 1),
             "ERRDET": ("buildErrorDetectPropS", ["t"], 1),
             "HYPER": ("buildHypercodeProperty", ["a.Sigma"], 1),
             "INALT": ("buildIATPropS", ["t"], 1),
             "INFIX": ("buildInfixProperty", ["a.Sigma"], 1),
```

```
            "OUTFIX": ("buildOutfixProperty", ["a.Sigma"], 1),
            "PREFIX": ("buildPrefixProperty", ["a.Sigma"], 1),
            "SUFFIX": ("buildSuffixProperty", ["a.Sigma"], 1),
            "TRAJECT": ("buildTrajPropS", ["$strexp", "$sigma"], 1)
}

tests = {"MAXP": "maximalP",
         "MAXW": "notMaximalW",
         "SATP": "satisfiesP",
         "SATW": "notSatisfiesW"}.
```

Next we show the actual program generation function which takes as parameters the property name (`pname`), which must be one appearing in the dictionary `buildName`, the question to answer (`test`), the file name for the automaton (`aname`), the possible trajectory regular expression string and alphabet (`strexp` and `sigma`), and the possible file name for the transducer (`tname`). The function returns a list of strings that constitute the lines of the desired Python program—we have omitted here the initial lines that contain the commands to import the required FAdo modules.

```
01   def program(pname, test=None, aname=None, strexp=None, sigma=None,
                 tname=None):
02       def expand(s):
03           s1 = Template(s)
04           return s1.substitute(strexp=strexp, sigma=sigma)

05       l = list()
06       l.append("ax = \"%s\"\n" % base64.b64encode(aname))
07       l.append("a = readOneFromString(base64.b64decode(ax))\n")
08       if buildName[pname][2] == 1:
09           if tname:
10               l.append("tx = \"%s\"\n" % base64.b64encode(tname))
11               l.append("t = base64.b64decode(tx)\n")
12           s = "p = " + buildName[pname][0] + "("
13           for s1 in buildName[pname][1]:
14               if s1 == "$strexp":
15                   s += "t, "
16               else:
17                   s += "%s," % expand(s1)
18           s = s[:-1] + ")\n"
19           l.append(s)
20           l.append("print p.%s(a)\n" % tests[test])
21       else: ..............
```

We refer to the lines above as meta-lines, as they are used to generate lines of the desired Python program. Meta-line 06 above generates the first line of the program, which would read the automaton file in a string `ax` that is encoded in binary to allow for safe transmission and reception over different operating systems. The next meta-line generates the line that would create an NFA object from the decoded string `ax`. The if part of the `if-else` statement is the one we use in this paper—the else part is for other LaSer questions. If there is a transducer file then, as in the previous case of the automaton file, LaSer generates a line that would create an SFT object from the encoded file. Then, meta-lines 12–18 generate a line that would create the property `p` to be processed, using the appropriate build-property function. Finally, meta-line 20 generates the line that would print the result of invoking the desired method `test` of the property `p`.

9 Concluding Remarks

We have presented a simple to use set of methods and functions that allow one to define many natural code properties and obtain answers to the satisfaction and maximality problems with respect to these properties. This capability relies on our implementation of basic transducer methods, including our witness version of the non-functionality transducer method, in the FAdo set of Python packages. We have also produced a new version of LaSer that allows clients to inquire about error-detecting and -correcting properties, as well as to generate programs that can be executed and provide answers at the client site.

There are a few important directions for future research. First, the existing implementation of transducer objects is not always efficient when it comes to describing code properties. For example, the transducer defined in Example 11 consists of 6 transitions. In general, if the alphabet is of size s, then that transducer would require $s + s(s-1) + s = s^2 + s$ transitions. However, a symbolic notation for transitions, say of the form

```
0 @s @s 0
0 @s @s' 1
1 @s @s 1
```

is more compact and can possibly be used by modifying the appropriate transducer methods. In this hypothetical notation, `@s` represents any alphabet symbol and `@s'` represents any alphabet symbol other than the previous one. Of course the syntax of such transducer descriptions has to be defined carefully so as to be as expressive as possible and at the same time efficiently utilizable in transducer methods. We note that a similar idea for symbolic transducers is already investigated in [35].

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 the satisfaction problem. Moreover, these methods should be capable of allowing to express properties that cannot be expressed in the transducer methods considered here. In particular, all transducer properties in this work are independences with parameter $n = 3$, so they do not include, for instance, the comma-free code property. A language L is a comma-free code [31] if

$$LL \cap \Sigma^+ L \Sigma^+ = \emptyset.$$

The formal method of [14] is quite expressive, using a certain type of 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 defining method is too expressive then even the satisfaction problem could become undecidable—see for example the method of multiple sets of trajectories in [8].

We consider the construction problem to be the Holy Grail of coding theory. We believe that as long as the satisfaction problem is efficiently decidable one can use an algorithmic approach to address the problem to some extent. For example, we already have implemented a statistical algorithm that starts with an initial small language L (in fact a singleton one) and then performs a loop in which it does the following task. It randomly picks words w of length ℓ that are not in L until $L \cup w$ satisfies the property, in which case L becomes $L \cup w$, or no such w is found after a MAX value of trials. The loop terminates either when L contains n words (as requested), or when no new word can be added after MAX trials. The MAX value is defined such that the statistical confidence about the output is quite high—in practice, this means that if the algorithm terminates with a language L containing less than n words, then the ratio, over $|\Sigma|^\ell$, of the number of words that could possibly be added to L is very small. Although this is a simple approach, we feel that it can lead to further developments with respect to the construction problem.

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 January, 2015.

[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 [27], 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] LaSer. Independent LAnguage SERver. URL address: http://laser.cs.smu.ca/independence/ Accessed in January, 2015.

[18] 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.

[19] OpenFst Library. Google Research and NYU's Courant Institute. URL address: http://www.openfst.org/ Accessed in January, 2015.

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

[21] Alexandru Mateescu and Arto Salomaa. Formal languages: an introduction and a synopsis. In Rozenberg and Salomaa [27], pages 1–39.

[22] 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.

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

[24] 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.

[25] Python. The official home of the python programming language. URL address: https://www.python.org/ Accessed in September of 2014.

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

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

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

[29] 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.

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

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

[32] 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.

[33] Western University. Grail+. URL address: http://www.csit.upei.ca/~ccampeanu/Grail/ Accessed in January, 2015.

[34] Vaucanson. The vaucanson project. URL address: http://vaucanson-project.org/ Accessed in September, 2014.

[35] 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.

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

[37] Sheng Yu. Regular languages. In Rozenberg and Salomaa [27], pages 41–110.

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