Implementation of Code Properties via Transducers* **

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

[1] 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, 4169-007 Porto Portugal {nam,rvr}@dcc.fc.up.pt

Abstract. The FAdo system is a symbolic manipulator of formal language 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 one to query about a code property and obtain the answer in a batch mode. Our work is founded on independence theory as well as the theory of rational relations and transducers, and contributes with improved algorithms on these objects.

Keywords: automata, codes, FAdo, implementation, language properties, regular languages, symbolic computation, transducers, program generation

1 Introduction

Several programming platforms are nowadays available, providing methods to transform and manipulate various formal language objects: Grail/Grail+ [10,24], Vaucanson 1 [5], Vaucanson-R [30], FAdo [1,9], JFLAP and OpenFST [22]. 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

* Due to the page limit we chose to omit algorithmic details and proofs of correctness, and focus on providing a somewhat comprehensive presentation on implementation aspects and the new capabilities of FAdo. Details can be found in [17].

** N. Moreira and R. Reis are partially supported by CMUP (UID/MAT/00144/2013), which is funded by FCT with national and European structural funds through the programs FEDER, under the partnership agreement PT2020. S. Konstantinidis and C. Meijer are supported by NSERC, Canada.

shell. 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 system representations, etc. Due to the combinatorial nature of formal language representations, their calculations are almost impossible without computational aid.

In this work, we extend the capabilities of FAdo and LaSer [8, 19] 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. **(a)** Implementation of transducer objects and several transducer methods (various product constructions, rational operations, transducer functionality test) *(Sect. 3)*. **(b)** 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 addition to some fixed code properties (such as prefix code, infix code, hypercode), these methods can be used to construct new code properties and combine existing properties, including various error-detecting properties *(Sect. 4)*. **(c)** Enhancement and implementation of decision algorithms for code properties of regular languages. In particular, such algorithms have been implemented and enhanced so as to provide witnesses in case of a negative answer *(Sect. 5)*. To our knowledge such implementations are not openly available. **(d)** 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 *(Sect. 4)*. **(e)** Generation of executable Python code based on the requested question about a given code property. This is mostly of use in the online LaSer, which receives client requests and attempts to compute answers *(Sect. 6)*. **(f)** All the above classes and methods are open source (GPL). Our work is founded on independence theory [15, 29] as well as the theory of rational relations and transducers [3, 26].

2 Terminology and Background

Sets, alphabets, words, languages. If S is a set, then $|S|$ denotes the cardinality of S, and 2^S denotes the set of all subsets of S. An *alphabet* is a finite nonempty set of symbols. In this paper, we write Σ, Δ for any arbitrary alphabets. The set of all words, or strings, over an alphabet Σ is written as Σ^*, which includes the *empty word* ε. A *language* (over Σ) is any set of words. We use standard operations and notation on words and languages [13, 20, 25].

Codes, properties, independent languages. A *code property*, or *independence*, [15], is a set $\mathcal{P}$ of languages for which there is $n \in \mathbb{N} \cup \{\aleph_0\}$ such that $L \in \mathcal{P}$, if and only if $L' \in \mathcal{P}$, for all $L' \subseteq L$ with $0 < |L'| < n$. If L is in $\mathcal{P}$ then we say that L *satisfies* $\mathcal{P}$. Thus, L satisfies $\mathcal{P}$ exactly when all nonempty subsets of L with less than n elements satisfy $\mathcal{P}$. 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 $w \notin L$. We note that every

L satisfying $\mathcal{P}$ is included in a maximal $\mathcal{P}$ code [15]. As far as we know, all code related properties in the literature [4,6,8,11,15,23,28] are code properties as defined here. The focus of this work is on 3-independences that can also be viewed as independences with respect to a binary relation in the sense of [29].

Automata [26,32]. A nondeterministic finite automaton with ε-transitions, for short *automaton* or *ε-NFA*, is a quintuple $\mathbf{a} = (Q, \Sigma, T, I, F)$ such that Q is the finite 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*. 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 ε, that is, $T \subseteq Q \times \Sigma \times Q$.

Transducers and (word) relations [3,26,32]. 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 by 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 *size* of $(p, x/y, q)$ is the number $1 + |x| + |y|$. The size $|\mathbf{t}|$ of $\mathbf{t}$ is the sum of $|Q|$ and the sizes of all transitions. The relation $\mathrm{R}(\mathbf{t})$ *realized by* the transducer $\mathbf{t}$ 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 $\mathbf{t}^{-1}$, 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 $y \in (\Delta \cup \varepsilon)$. If $\mathbf{s}$ and $\mathbf{t}$ are transducers, then there is a transducer $\mathbf{s} \vee \mathbf{t}$ realizing $\mathrm{R}(\mathbf{s}) \cup \mathrm{R}(\mathbf{t})$.

3 Transducer Object Classes and Methods

Here we discuss some aspects of the implementation of transducer objects and related methods. These are contained in the module `transducers.py` and can be imported as follows:

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

The FAdo class `GFT`, for General Form Transducer, is a subclass of `NFA`, which is the FAdo class for ε-NFAs. 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 Python 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 the method `toSFT()`.

Example 1. The following code defines a string `s` containing a transducer description, and then constructs an SFT transducer `t` from `s` via a method of the module `fio`, which contains input/output methods for formal language objects. On input x, `t` returns the set of all proper suffixes of x—see also Fig 1.[3] It has an initial state 0 and a final state 1, and deletes at least one of the input symbols.

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

As usual, `\n` denotes the *end of line character*, so the string `s` consists of 7 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 $(0, \texttt{a}/\varepsilon, 0)$; the last line contains the transition $(1, \texttt{b}/\texttt{b}, 1)$. Here `t.Sigma={a,b}`.

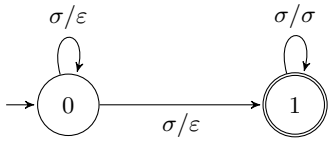

Fig. 1. On input x, the above transducer outputs any proper suffix of x.

Recall, for a transducer $\mathbf{t}$ and word w, $\mathbf{t}(w)$ is the set of possible outputs of $\mathbf{t}$ on input w. Note that this set can 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 2. The following code is a continuation of Ex. 1. It prints the set of all proper suffixes of the word `ababb`, which are all of length ≤ 4.

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

[3] Note: In 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 each alphabet symbol σ; and similarly for an arrow with label σ/ε. An arrow with label σ/σ' represents a set of transitions with labels σ/σ' for all distinct alphabet symbols σ, σ'.

Assuming `t` is an SFT object, the following methods are available: "`t.inverse()`" returns the inverse of `t`; "`t.evalWordP((x,y))`" returns `True` or `False`, depending whether the pair `(x,y)` belongs to the relation realized by `t`; "`t.nonEmptyW()`" returns some word pair `(x, y)` which belongs to the relation realized by `t`, if nonempty; otherwise, it returns the pair `(None, None)`.

Product constructions [3, 16, 32]. The next methods are adaptations of the standard product construction [13] between two NFAs which produces an NFA accepting the intersection of the corresponding languages. 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)`" as above except that y is accepted by `a` ; "`t.runOnNFA(a)`" returns the NFA accepting the language $\bigcup_{w \in \mathrm{L}(\mathtt{a})} \mathtt{t}(w)$; "`t.composition(s)`" returns a transducer realizing the composition $\mathrm{R}(\mathtt{t}) \circ \mathrm{R}(\mathtt{s})$.

Rational operations [3]. 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)`; `t.concat(s)`; `t.star()`. The implementation of these methods mimics the implementation of the corresponding methods on automata.

***Witness of Transducer* non-*functionality*.** A transducer $\mathtt{t}$ is called *functional* if $|\mathtt{t}(w)| \leq 1$, for every word w. Transducer functionality can be tested in polynomial time [2]. A triple of words (w, z, z') is called a *witness of* $\mathtt{t}$*'s non-functionality*, if $z \neq z'$ and $z, z' \in \mathtt{t}(w)$. We have implemented the SFT method `t.nonFunctionalW()`, which returns a witness of `t`'s non-functionality, or the triple `(None,None,None)` if `t` is functional. Our method is based on the decision test and uses extra bookkeeping for producing the desired witness.

Theorem 1. *The FAdo method* `t.nonFunctionalW()` *computes a size* $O(|\mathtt{t}|^2)$ *witness of* `t`*'s non-functionality, if and only if one exists.*

The proof of correctness can be found in [17]. There is a sequence $(\mathtt{t}_n)$ of transducers such that $|\mathtt{t}_n| \to \infty$ and the minimal witness of each $\mathtt{t}_n$ is of size $\Theta(|\mathtt{t}_n|^2)$.

4 Object Classes Representing Code Properties

In this section we discuss our implementation of objects representing code properties. We are interested in 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 [6], where a property is described by a regular expression over $\{0, 1\}$, and the transducers of [8], where a property is described by a transducer. These formal methods can 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.

Input-altering transducer properties [8]. 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 $\mathcal{P}^{al}_{\mathbf{t}}$ consisting of all languages L 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 Ex. 1 is input-altering and describes the suffix code property over the alphabet {`a, b`}.

Error-detecting properties via input-preserving transducers [8, 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 $\mathcal{P}^{ed}_{\mathbf{t}}$ consisting of all languages L over the input alphabet of $\mathbf{t}$ such that

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

Every input-altering transducer property is an error-detecting property [8].

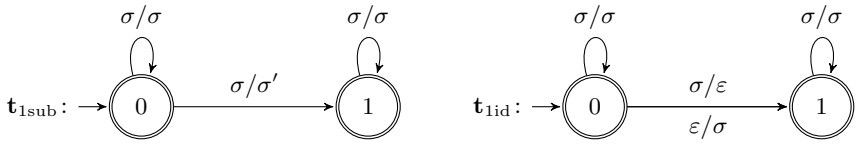

Fig. 2. On input x, $\mathbf{t}_{1\mathrm{sub}}$ outputs either x, or any word that results by substituting exactly one symbol in x. On input x, $\mathbf{t}_{1\mathrm{id}}$ outputs either x, or any word that results by deleting, or inserting, exactly one symbol in x.

Example 3. Consider the property *1-substitution error-detecting code* over {`a, b`}, where error means the substitution of one symbol by another symbol. The following channel transducer defines this property—see also Fig 2. The transducer will substitute at most one symbol of the input word with another symbol.

```
s1   = '@Transducer 0 1 * 0\n0 a a 0\n0 b b 0\n'\
          '0 b a 1\n0 a b 1\n1 a a 1\n1 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 2. Codes for such errors are actively under investigation—see [23], for instance.

4.1 Implementation in FAdo.

We have defined the Python classes `TrajProp, IATProp` and `ErrDetectProp` corresponding to the types of properties discussed above. These property types are described, respectively, by regular trajectory expressions, input-altering transducers, and input-preserving transducers. In all 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 4. Consider again Examples 1,3 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)
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 two statements, `scp, s1dp` represent, respectively, the suffix code property and the 1-substitution error-detecting property. The last two statements are explained below—`pcp` and `icp2` represent the prefix code and infix code properties, respectively.

Fixed properties. We have created specific classes for the well-known properties prefix, suffix, infix, outfix, and hypercodes. As before, users need only to know about the `build`-interfaces for creating objects of these classes. For example, `buildPrefixProperty(Sigma)` returns an object of the class `PrefixProp` that represents all prefix codes over the alphabet `Sigma`.

4.2 Combining code properties

In many cases it is desirable to talk about languages satisfying more than one property. For example, most of the practical 1-substitution error-detecting codes 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 5. Using the properties `icp, s1dp` created above in Ex. 4, 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` 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`.

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

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

In fact, we have also implemented the methods '`&`' and '$\leq$' in a way that the triple (`ErrDetectProp`, $\&, \leq$) constitutes a syntactic hierarchy (see further below). 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`; $\qquad$ `p` $\leq$ `q` if and only if `p & q` returns `p`

Our implementation associates to each `ErrDetectProp` object `p` a nonempty set `p`.ID of names. If `p` is a fixed property object, `p`.ID has one hardcoded name. If `p` is built from a transducer `t`, `p`.ID has one name, the name of `t`—this name is based on a string description of `t`. If `p` = `q&r`, then `p`.ID is the union of `q`.ID and `r`.ID minus any fixed property name N for which another fixed property name M exists in the union such that the M-property is contained in the N-property—see [17] for details.

Next we define what it means to simulate a set of code properties $\mathcal{Q} = \{\mathcal{Q}_j \mid j \in J\}$ via a syntactic hierarchy $(G, \&, \leq)$, which can ultimately be implemented (as is the case here) in a 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 $x, y \in \langle G\rangle$. We show that there is an efficient simulation of the set of properties $\mathcal{P}^{\text{ed}}$ in (3) and that there can be no *complete* simulation of that set of properties.

Definition 1. *A* syntactic hierarchy *is a triple $(G, \&, \leq)$ where G is a nonempty set and (a) $(\langle G\rangle, \&)$ is the commutative semigroup generated by G with computable operation '&'. (b) $(\langle G\rangle, \leq)$ is a decidable partial order (reflexive, transitive, antisymmetric). (c) For all $x, y \in \langle G\rangle$, we have that $x \leq y$ implies $x\&y = x$, and that $x\&y \leq x$.*

Definition 2. *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; $[\,] : \langle G \rangle \to \mathcal{Q}$ is a surjective mapping; $\varphi : J \to \langle G \rangle$ with $[\varphi(j)] = \mathcal{Q}_j$; for all $x, y \in \langle G \rangle$, $x \leq y$ implies $[x] \subseteq [y]$; and for all $x, y \in \langle G \rangle$, $[x\&y] = [x] \cap [y]$. The simulation is called* complete *if, for all x, y, $[x] \subseteq [y]$ implies $x \leq y$. 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 2. *There is a linear simulation of the set of properties $\mathcal{P}^{\mathrm{ed}}$.*

Theorem 3. *There is no complete simulation of the set of properties $\mathcal{P}^{\mathrm{ed}}$.*

The above result implies that for any FAdo `ErrDetectProp` objects `p, q` defined via transducers `t` and `s` with $\mathcal{P}^{ed}_{\mathtt{t}} \subseteq \mathcal{P}^{ed}_{\mathtt{s}}$ 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. Using the notation of Ex. 4, this implies that

```
pcp & icp2  returns  icp2
```

5 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 the *witness version* of this problem, a negative answer is also accompanied by a 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 `t` and, in the first two problems, the language is given via an NFA `a`. Next we discuss the implementation of methods for the satisfaction problem, all of which work in polynomial time. Due to the page limit we omit details on the maximality problems. Aspects of the construction problem are discussed in [18].

Methods `p.satisfiesP(a)`. Eq. (1) implies that, if the property `p` is described by an input-altering transducer `t`, the method `p.satisfiesP(a)` can be implemented as follows, where `&` is NFA intersection

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

If `p` is an error-detecting property, the transducer `t` is input-preserving and Eq. (2) is tested via transducer functionality. In FAdo this test can be done as follows, where `functionalP()` returns whether a transducer is functional.

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

Methods with witnesses: `p.notSatisfiesW(a)`. For input-altering transducer and error-detecting properties, the witness version of `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 $\mathrm{L}(\mathtt{a})$. We accomplish this by changing appropriately the implementations of `p.satisfiesP(a)` shown before—see [17] for details.

Example 6. The following Python interaction shows that a^*b is a prefix and 1-error-detecting code. The strings `st, s1` contain the descriptions of an NFA accepting a^*b, and a transducer allowing up to 1 substitution error on the input word.

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

Uniquely Decipherable Codes. The property of unique 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 [27] as well as formal languages [21]. This property is not defined via a transducer and is treated differently. In particular the witness version of the satisfaction problem is solved based on the decision algorithm of [12]—see [17] for details. Next we only show an example of how one can use the satisfaction method.

Example 7. The following Python interaction produces a witness of the non-satisfaction of the UD code property by the finite language $L = \{ab, abba, bab\}$.

```
>>> 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 their concatenations form equal words.

6 LaSer and Program Generation

The first version of LaSer [8] was a limited and self-contained set of C++ automaton and transducer methods with a web interface having the following functionality: a user uploads a file containing an automaton 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 discussed 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. A 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, can be quite time consuming for a server software.

7 Concluding Remarks

There are a few directions for future research. First, the existing implementation of transducers is not always efficient when it comes to describing code properties. For example, the transducer defined in Ex. 3 consists of 6 transitions. In general, if the alphabet has size s, then that transducer would require $s+s(s-1)+s = s^2+s$ transitions. However, a symbolic notation for transitions would be more compact and can possibly be used by modifying the appropriate transducer methods—certain symbolic transducers are investigated in [31]. Formal methods for defining code properties need to be evolved further with the aim of ultimately implementing these properties and answering efficiently the satisfaction problem. These methods should be capable of allowing to express properties that cannot be expressed in the transducer methods. In particular, as all transducer properties in this work are 3-independences, they do not include properties like comma-free code property. The formal method of [14] is quite expressive, using a certain type of first order formulae to describe properties. 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 [7].

References

1. Almeida, A., Almeida, M., Alves, J., Moreira, N., Reis, R.: FAdo and GUItar: Tools for automata manipulation and visualization. In: Proceedings of 14th CIAA 2009. LNCS, vol. 5642, pp. 65–74. Springer (2009)
2. Béal, M.P., Carton, O., Prieur, C., Sakarovitch, J.: Squaring transducers: An efficient procedure for deciding functionality and sequentiality. Theoretical Computer Science 292(1), 45–63 (2003)
3. Berstel, J.: Transductions and Context-Free Languages. B.G. Teubner (1979)
4. Berstel, J., Perrin, D., Reutenauer, C.: Codes and Automata. Cambridge University Press (2009)

5. Claveirole, T., Lombardy, S., O'Connor, S., Pouchet, L., Sakarovitch, J.: Inside vaucanson. In: Proceedings 10th CIAA 2005. LNCS, vol. 3845, pp. 116–128. Springer (2005)
6. Domaratzki, M.: Trajectory-based codes. Acta Informatica 40, 491–527 (2004)
7. Domaratzki, M., Salomaa, K.: Codes defined by multiple sets of trajectories. Theoretical Computer Science 366, 182–193 (2006)
8. Dudzinski, K., Konstantinidis, S.: Formal descriptions of code properties: decidability, complexity, implementation. IJFCS 23:1, 67–85 (2012)
9. FAdo: Tools for formal languages manipulation, http://fado.dcc.fc.up.pt/
10. Grail: Grail+, http://www.csit.upei.ca/~ccampeanu/Grail/
11. Hamming, R.W.: Error detecting and error correcting codes. The Bell System Technical Journal 26(2), 147–160 (1950)
12. Head, T., Weber, A.: Deciding code related properties by means of finite transducers. In: Sequences II, Methods in Communication, Security, and Computer Science. pp. 260–272. Springer-Verlag (1993)
13. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
14. Jürgensen, H.: Syntactic monoids of codes. Acta Cybernetica 14, 117–133 (1999)
15. Jürgensen, H., Konstantinidis, S.: Codes. In: Rozenberg and Salomaa [25], pp. 511–607
16. Konstantinidis, S.: Transducers and the properties of error-detection, error-correction and finite-delay decodability. JUCS 8, 278–291 (2002)
17. Konstantinidis, S., Meijer, C., Moreira, N., Reis, R.: Symbolic manipulation of code properties. Computing Research Repository (2015), arXiv:1504.04715v1
18. Konstantinidis, S., Moreira, N., Reis, R.: Channels with synchronization/substitution errors and computation of error control codes. Computing Research Repository (2016), arXiv:1601.06312v1
19. LaSer: Independent LAnguage SERver, http://laser.cs.smu.ca/independence/
20. Mateescu, A., Salomaa, A.: Formal languages: an introduction and a synopsis. In: Rozenberg and Salomaa [25], pp. 1–39
21. Nivat, M.: Elements de la théorie générale des codés. In: Automata Theory, pp. 278–294 (1966)
22. OpenFst: OpenFst Library, http://www.openfst.org/
23. Paluncic, F., Abdel-Ghaffar, K., Ferreira, H.: Insertion/deletion detecting codes and the boundary problem. IEEE Trans. Info. Theory 59(9), 5935–5943 (2013)
24. Raymond, D., Wood, D.: Grail: A C++ library for automata and expressions. Journal of Symbolic Computation 17(4), 341 – 350 (1994)
25. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, Vol. I. Springer-Verlag, Berlin (1997)
26. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press (2009)
27. Sardinas, A.A., Patterson, G.W.: A necessary and sufficient condition for the unique decomposition of coded messages. IRE Int. Conven. Rec. 8, 104–108 (1953)
28. Shyr, H.J.: Free Monoids and Languages. Hon Min Book Company, Taichung, second edn. (1991)
29. Shyr, H.J., Thierrin, G.: Codes and binary relations. In: Malliavin, M.P. (ed.) Séminaire d'Algèbre Paul Dubreil, Paris 1975–1976 (29ème Année). LNCS, vol. 586, pp. 180–188. Springer (1977)
30. Vaucanson: The Vaucanson Project, http://vaucanson-project.org/
31. Veanes, M.: Applications of symbolic finite automata. In: Proceedings of 18th CIAA 2013. LNCS, vol. 7982, pp. 16–23. Springer (2013)
32. Yu, S.: Regular languages. In: Rozenberg and Salomaa [25], pp. 41–110