International Journal of Foundations of Computer Science

FORMAL DESCRIPTIONS OF CODE PROPERTIES: DECIDABILITY, COMPLEXITY, IMPLEMENTATION*

KRYSTIAN DUDZINSKI and STAVROS KONSTANTINIDIS

Department of Mathematics and Computing Science, Saint Mary's University, Halifax, Nova Scotia, B3H 3C3 Canada.
krystian_dudzinski@hotmail.com, s.konstantinidis@smu.ca

Received (Day Month Year)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

The branch of coding theory that is based on formal languages has produced several methods for defining code properties, including word relations, dependence systems, implicational conditions, trajectories, and language inequations. Of those, the latter three can be viewed as *formal* methods in the sense that a certain formal expression can be used to denote a code property. Here we present a formal method which is based on transducers. Each transducer of a certain type defines/describes a desired code property. The method provides simple and uniform decision procedures for the basic questions of property satisfaction and maximality for regular languages. Our work includes statements about the hardness of deciding some of the problems involved. It turns out that maximality can be hard to decide even for "classical" code properties of *finite* languages. We also present an initial implementation of a LAnguage SERver capable of deciding the satisfaction problem for a given transducer code property and regular language.

Keywords: automata, codes, complexity, decidability, descriptions, implementation, languages, maximality, properties, transducers.

1991 Mathematics Subject Classification: 68Q45, 68Q25, 94A45

1. Introduction

The branch of coding theory that is based on formal languages [4, 39, 20, 43] has matured to a point that allows one to express, mathematically, a wide variety of code properties. In particular, several methods now exist for defining code properties. These include word relations [40, 39, 22, 44], (in)dependence systems [23, 20], implicational conditions [18], trajectory sets [7, 8], and language inequations [24]. Of these, the latter three can be viewed as *formal* or close to formal methods in the sense that a certain type of formal expression can be used to define/describe a code property.

*Research supported by NSERC.

In this work we present a formal method which is based on transducers: each transducer of a certain type describes a desired code property. This method can be viewed as a formal approach to the binary word relations and inequations methods, in which we employ transducer and automaton techniques to decide code properties. It provides a simple mechanism for describing code properties, as well as *simple and uniform* algorithms for the basic decision problems of property satisfaction and maximality for regular languages. Our work includes statements about the hardness of deciding these problems and shows, in particular, that the property maximality problem can be hard. We note that, unlike the property satisfaction problem, which has been investigated considerably – e.g., [38, 33, 31, 20, 11] – the property maximality problem has been addressed systematically only recently [7, 24, 44] – for more specific and deep contributions see [28, 29]. Our transducer methodology has led to the partial development and implementation of a LAnguage SERver [30] which is currently capable of deciding the satisfaction problem for a given code property and regular language. In particular, via a web application, a user can enter two files containing the description of the desired property (via a transducer or a regular trajectory set) and the description of a regular language (via an NFA: nondeterministic finite automaton), and the server decides whether the language satisfies the property.

The paper is organized as follows. The next section contains the basic notation and terminology from automata, languages, and codes. Section 3 gives a brief overview of existing methods defining code properties. In that section, we choose the independence method as the most natural one for defining code properties in the broad sense of the term, and we show that there are uncountably many independence properties whose elements are infix codes. Then, we focus on existing *formal* methods. In Section 4, we present our objectives in defining transducer based formal methods. In particular, we consider two types of properties. The first one is based on input-altering transducers, and the second one is based on input-preserving transducers. We relate these to some of the existing methods and present algorithms for deciding the property satisfaction and maximality problems with our method. As corollaries we obtain that the maximality problem for the properties of thinness, error-detection and -correction is decidable. While the satisfaction problem is polynomially decidable, the maximality problem is shown to be PSPACE-hard in general. Section 5 follows up on the hardness of the maximality problem and focuses on some fixed "classical" code properties for finite languages. It is shown that then the maximality problem is coNP-hard. In Section 6, we discuss some aspects of the functionality, user interface, and architecture of the language server. Finally, in the last section we summarize our contributions and discuss possible future directions.

2. Basic notation and background

2.1. *Sets, words, iff, languages, properties, relations*

If S is any set, the expressions $|S|$ and 2^S denote, respectively, the cardinality and the power set of S. When there is no risk of confusion we denote a singleton set $\{u\}$ simply as u. For example, $S \cup u$ is the union of S and $\{u\}$. We use the standard basic notation and terminology for alphabets, words and languages – see [35], [42]. For example, Σ denotes an alphabet, Σ^+ the set of nonempty words, λ the empty word, $|w|$ the length of the word w. We use the concepts of (formal) language and concatenation between words, or languages, in the usual way. We say that w is an L-word if $w \in L$ and L is a language. The notation L_λ is shorthand for $L \cup \lambda$. The acronym *iff* stands for "if and only if". A *language property* is any set of languages, that is, a subset of 2^{Σ^*}. If $\mathcal{P}$ is a language property and $L \in \mathcal{P}$ then we say that L *satisfies* $\mathcal{P}$, or *has* the property $\mathcal{P}$. If in addition there is no proper superset of L satisfying $\mathcal{P}$, then L is called a $\mathcal{P}$-*maximal* language, or a maximal $\mathcal{P}$ language. A binary word *relation* ρ on Σ^* is any subset of $\Sigma^* \times \Sigma^*$. The *domain* of ρ is $\{u \mid (u, v) \in \rho \text{ for some } v \in \Sigma^*\}$. The inverse of ρ is the relation $\rho^{-1} = \{(v, u) \mid (u, v) \in \rho\}$.

2.2. *Code properties*

What constitutes a code property (or code-related property) is a matter of debate. Here we take the approach of dependence systems [23, 20] and define a *code property* to be any language property that is an independence (see the next section). Next we list a few code properties that we shall refer to in this work – see [4, 39, 20, 43] for information on "classical" code properties, and [16, 25] for DNA-related code properties. A language K is called a uniquely decodable code, or simply a *code*, if, for every word $w \in K^*$, there is a unique sequence of K-words whose concatenation is equal to w. We write $\mathcal{P}_{\text{code}}$ for the (uniquely decodable) code property. A language K is called a *prefix* code (resp., *suffix, infix* code) if K contains no two different words of the form u and ux (resp., xu, xuy). A language is called *thin* [36] if it contains no two different words of the same length – in [4] the term thin has a totally different meaning.

We note that, by identifying "code properties" with independence properties, we can have languages having independence properties without being (uniquely decodable) codes – e.g., thin languages like $\{a, aa\}$.

A *channel* γ is a binary relation on Σ^* that is domain-preserving (or input-preserving); that is, $\gamma \subseteq \Sigma^* \times \Sigma^*$ and $(w, w) \in \gamma$ for all words w in the domain of γ. When $(x, y) \in \gamma$ we say that the channel input x can result in, or be received as, y. A language L is *error-detecting for* γ, if no channel input $x \in L_\lambda$ can result in a different output $y \in L_\lambda$: $(x, y) \in \gamma \cap (L_\lambda \times L_\lambda)$ implies $y = x$. The language is *error-correcting for* γ, if no two different channel inputs in L_λ can result into the same output – see [26] for more details on these concepts.

2.3. *NFAs and transducers*

A nondeterministic finite automaton with empty transitions, λ-*NFA* for short, is a quintuple $\hat{a} = (Q, \Sigma, T, s, F)$ such that Q is the set of states, Σ is the alphabet, $s \in Q$ is the start (or initial) state, $F \subseteq Q$ is the set of final states, and $T \subseteq Q \times (\Sigma \cup \lambda) \times Q$ is the finite set of transitions. Let (p, x, q) be a transition of $\hat{a}$. Then x is called the *label* of the transition, and we say that p has an *outgoing* transition (with label x). A *path* of $\hat{a}$ 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 $\hat{a}$. The word $x_1 \cdots x_\ell$ is called the label of the path. The path is called *accepting* if p_0 is the start state and p_ℓ is a final state. The *language accepted* by $\hat{a}$, denoted as $L(\hat{a})$, is the set of labels of all the accepting paths of $\hat{a}$. The λ-NFA $\hat{a}$ is called *trim*, if every state appears in some accepting path of $\hat{a}$. The λ-NFA $\hat{a}$ is called an *NFA*, if no transition label is empty, that is, $T \subseteq Q \times \Sigma \times Q$. A deterministic finite automaton, *DFA* for short, is a special type of NFA where there is no state p having two outgoing transitions with different labels.

A (finite) *transducer* [3] is a sextuple $\hat{t} = (Q, \Sigma, \Delta, T, s, F)$ such that Q, s, F are exactly the same as those in λ-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 λ-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 $\hat{t}$, denoted as $R(\hat{t})$, is the set of labels in all the accepting paths of $\hat{t}$. The transducer $\hat{t}$ is said to be in *standard form*, if each transition $(p, x/y, q)$ is such that $x \in (\Sigma \cup \lambda)$ and $y \in (\Delta \cup \lambda)$. We note that every transducer is effectively equivalent to one (realizing the same relation, that is) in standard form. We write $\hat{t}(x)$ for the set of possible outputs of $\hat{t}$ on input x, that is, $y \in \hat{t}(x)$ iff $(x, y) \in R(\hat{t})$.

3. Methods for defining code properties

The first mathematical method for studying code properties appears to be [40], and uses the concept of binary word relation. This approach led to further research developments – see e.g., [39, 43, 44]. In particular, for a binary word relation ρ, a language L is called ρ-*independent* if no two distinct elements of L are related with ρ. Thus, the relation ρ defines the property consisting of all ρ-independent languages. For example, the prefix relation ρ_{p}, with $(u, v) \in \rho_{\mathrm{p}}$ iff u is a prefix of v, defines the property of prefix codes. In [40], the authors consider binary word relations satisfying certain natural constraints, and show that the property $\mathcal{P}_{\mathrm{code}}$ is not definable by any such binary relation. In [22], the method of binary word relations is generalized naturally to n-ary word relations ω, where n is a positive integer. A language L is called ω-independent, if no tuple consisting of L-words is in ω. Under certain natural constraints on the allowable relations ω, it is shown that

the property $\mathcal{P}_{\text{code}}$ is not definable by any such relation.

In our opinion, the most natural and, at the same time, general method for defining code properties is the dependence systems[Σfootnote] of [23] – see also [20]. For brevity we skip dependence itself and go directly to the concept of independence. Let n be a positive integer, or $\aleph_0$. A property $\mathcal{P}$ is called an *n-independence* if

$$L \in \mathcal{P} \quad \text{iff} \quad \forall\, L' \subseteq L :\ 0 < |L'| < n \rightarrow L' \in \mathcal{P},$$

that is, L satisfies $\mathcal{P}$ exactly when every nonempty subset of L of cardinality less than n satisfies $\mathcal{P}$. We also call $\mathcal{P}$ an *independence* property if it is an n-independence for some n as above. It turns out that all properties considered in this paper are independence properties. This fact is significant because then the concept of maximality is well-defined:

Theorem 1. [19] *Let $\mathcal{P}$ be an independence property. Every language in $\mathcal{P}$ is included in a maximal $\mathcal{P}$ language.*

In the next subsection we discuss the direction of some authors towards formal methods.

3.1. *Formal methods for describing code properties*

While the previously discussed methods have been defined in rigorous mathematical terms, they do not provide any obvious method to *formally* define/describe code properties – in the sense that each of these properties is described via a formal expression. Three formal, or close to formal, methods have been proposed in the last decade or so.

Implicational conditions [18] In this method, code properties are described via first order formulae of a certain syntax, which are called implicational conditions. We refer the reader to [18] for details. Here we only show an implicational condition describing the suffix property:

$$\varphi_{\text{s}} = \text{“}\forall\, u, v, x :\ u \in L,\, v \in L,\, u = xv \ \rightarrow\ x = \lambda\text{”}.$$

The formula describes all suffix codes because it is satisfied exactly when the language L is a suffix code.

Regular Trajectories [7][Σfootnote] This is probably the simplest and most completely defined formal method. A regular trajectory property is described via a

[Σfootnote] To prevent misconceptions, we note that previous methods have played a fundamental role in our understanding of code properties, and we believe that they contributed to the maturity that was required in the research community to propose further methods. Moreover, the method of partial word orders continues to be of importance as it provides appropriate tools for addressing certain important problems, such as the embedding problem for code properties [44], whereas dependence systems appear to be too general for *these* problems.

[Σfootnote] In fact, [7] allows trajectory properties for which the defining set of trajectories is not necessarily regular. Here, however, we focus on methods with the constraints (or objectives) listed in Section 4.

regular expression $\bar{e}$ over the alphabet $\{0,1\}$ – in [7], words over this alphabet are called *trajectories* – such that a language L has the desired property if

$$(L \amalg_{\bar{e}} \Sigma^+) \cap L = \emptyset. \tag{1}$$

For example, the regular expression 1^*0^* describes the suffix property and $1^*0^*1^*$ describes the infix property. Here, $\amalg_h$ is the shuffle operation on the trajectory h. For any words u, w, $(u \amalg_h w)$ is the word of length $|h| = |u| + |w|$ resulting by shuffling the two words according to h: 0's in h mean to use letters from u and 1's in h mean to use letters from w – if this is not possible then $(u \amalg_h w) = \emptyset$. This operation extends naturally to languages, so $(L \amalg_{\bar{e}} \Sigma^+)$ is the union of $(u \amalg_h w)$, for all $u \in L, h \in L(\bar{e}), w \in \Sigma^+$ – see [34, 7] for more details.

Language inequations [24] This method defines a code property via a binary word operation $\diamond$ and a language R as the set of all languages L satisfying the inequation $(L \diamond R) \subseteq L^c$. Usually the fixed language R is equal to Σ^+. For example, using the word operation of left quotient, $\rightarrow_{\text{lq}}$, the inequation $(L \rightarrow_{\text{lq}} \Sigma^+) \subseteq L^c$ defines the suffix codes. Here, $v \in (u \rightarrow_{\text{lq}} x)$ iff v results by removing the prefix x from u, or equivalently, $u = xv$. As discussed in [24], some combinations of $\diamond$ and R can be described via a transducer $[\diamond R]$ such that $v \in [\diamond R](u)$ iff $v \in u \diamond R$. In the next section we follow-up on this idea and propose a formal method that is based solely on transducers.

In a formal method, like the ones discussed above or in the next section, one of the objectives is to have a well-defined set of formal expressions (descriptions), each of which defines a language property. Thus, any formal method can describe countably many properties. On the other hand, as shown next, the set of independence properties is uncountable and, therefore, we cannot expect to formally describe all possible code properties. One might protest that the independence method allows one to define too many properties – for instance, thin languages that are not codes in the sense of unique decodability. It turns out, however, that the situation will not change even if we restrict our attention to independence properties containing only *infix* codes:

Theorem 2. *Let n be either $\aleph_0$, or an integer greater than 1. There are uncountably many n-independence properties whose elements are infix codes. Hence, also there are uncountably many n-independence properties.*

Proof. The proof relies on the following two claims.

Claim 1 If L is a language, then 2^L is an n-independence property.

Claim 2 There are uncountably many infix codes.

To see how the statement follows from these claims, let $\{C_i \mid i \in I\}$ be the family of all infix codes, where I is an index set. As each C_i is an infix code, the property 2^{C_i} consists of infix codes and, by the first claim, it is an n-independence property. Moreover, if $C_j \neq C_i$, then $2^{C_i} \neq 2^{C_j}$. Therefore, by the second claim, $\{2^{C_i} \mid i \in I\}$ is uncountable. The proofs of the claims are left to the reader – see also [9]. □

4. Transducer based properties

Here we propose transducer based formal methods with the following objectives in mind.

- As much as possible, we wish to be compatible with other existing methods.
- We wish to decide efficiently the *property satisfaction* problem: given the *description* of a code property $\mathcal{P}$ and the description of a regular language L, decide whether L satisfies $\mathcal{P}$.
- We wish to decide the *property maximality* problem: given the *description* of a code property $\mathcal{P}$ and the description of a regular language L, decide whether L is $\mathcal{P}$-maximal.
- We wish to build a LAnguage SERver allowing users to enter descriptions of code properties and produce answers to questions about languages with the desired code properties.

We note that the regular trajectory method [7] addresses successfully both the property satisfaction and maximality problems. Although the transducer methods cannot describe certain important properties, like $\mathcal{P}_{\rm code}$, they have several advantages: First, transducers are well understood objects in the formal language theory community and have a simple description: start state, list of final states, list of transitions. Second, the regular trajectories method is included in the transducer methods. Third, one can utilize directly the abundant algorithmic constructions on transducers and automata to obtain simple algorithms for the decision problems involved. *To avoid misconceptions, we emphasize that this work does not intend to define the "best" or "most general" method, but rather to address the objectives listed above, and to help bridge the gap between mathematical methods and implemented systems.*

The definitions of prefix, suffix and infix codes can be rephrased, respectively, as follows

$$\mathrm{pp}(L) \cap L = \emptyset, \;\; \mathrm{ps}(L) \cap L = \emptyset, \;\; \mathrm{pi}(L) \cap L = \emptyset, \tag{2}$$

where the operators $\mathrm{pp}, \mathrm{ps}, \mathrm{pi}$ return respectively, the *proper* prefixes, suffixes, and infixes of L. We note that, on each word u, these operators return a set of words *not* containing u. Moreover, each of these operators can be realized by a transducer. Next we generalize this observation by defining input-altering transducers and then we define the method of input-altering transducer properties. We assume that we have agreed on a fixed set $\mathbf{M}$ of words (our *maximum set*) such that all languages of interest are subsets of $\mathbf{M}$. The most common value for $\mathbf{M}$ in the literature on variable-length codes is Σ^*, but other values, such as Σ^n for some positive integer n, are considered (e.g., in the area of error-control codes). In any case, we assume that $\mathbf{M}$ is given via an automaton $\hat{m}$.

Definition 3. *A transducer $\hat{t}$ is called* input-altering *if*

$$w \notin \hat{t}(w), \text{ for all } w \in \mathbf{M}. \tag{3}$$

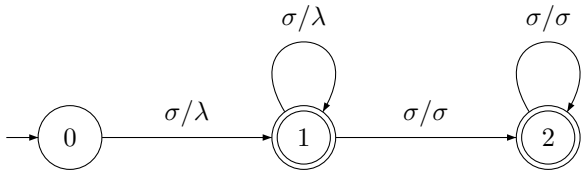

Fig. 1. Input-altering transducer describing the "suffix" property. Each arrow represents multiple transitions (one for each alphabet symbol σ). On input u, the transducer returns one of the possible *proper* suffixes of u.

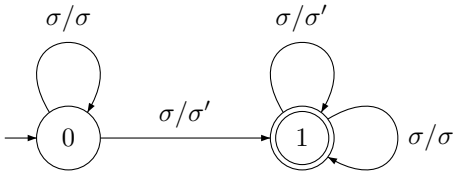

Fig. 2. Input-altering transducer describing the "thin" property. Again, each arrow represents multiple transitions (σ/σ' means one label for each combination of distinct alphabet symbols σ, σ'). On input u, the transducer returns one of the possible words that have the same length with u and differ from u in at least one position.

A language property is an input-altering transducer property *if there is an input-altering transducer $\hat{t}$ such that the property consists of all languages L satisfying*

$$\hat{t}(L) \cap L = \emptyset. \tag{4}$$

We denote by $\mathcal{P}^{\mathrm{al}}_{\hat{t}}$ the property defined by the input-altering transducer $\hat{t}$.

Fig. 1 shows an input-altering transducer realizing the operation "ps" shown in (2). Fig. 2 shows an input-altering transducer for the "thin" property.

It appears that the properties of error-detection and -correction are not definable in the methodology of input-altering transducer properties. On the other hand, these properties can be defined naturally using the input-preserving type of transducers.

Definition 4. *A transducer $\hat{t}$ is called* input-preserving *if*

$$w \in \hat{t}(w), \text{ for all } w \in \mathbf{M}. \tag{5}$$

A language property is an input-preserving transducer property *if there is an input-preserving transducer $\hat{t}$ such that the property consists of all languages L satisfying*

$$\hat{t}(u) \cap (L - u) = \emptyset, \text{ for all } u \in L. \tag{6}$$

We denote by $\mathcal{P}^{\mathrm{pr}}_{\hat{t}}$ the property defined by the input-preserving transducer $\hat{t}$.

Fig. 3 shows an input-preserving transducer defining the suffix property – compare with Fig. 1. In Fig. 4 the input-preserving transducer defines the error-detection

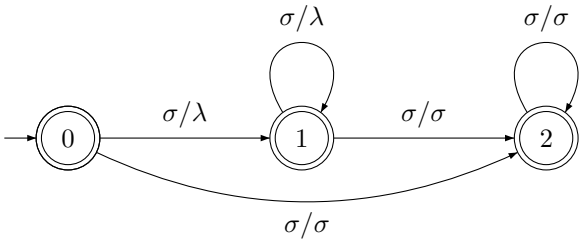

Fig. 3. Input-preserving transducer describing the "suffix" property. On input u, the transducer returns one of the possible suffixes of u, including u.

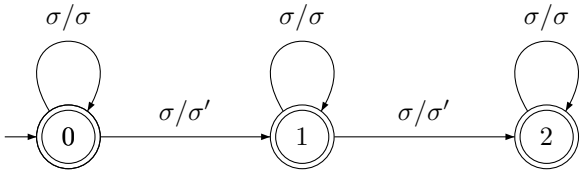

Fig. 4. Input-preserving transducer describing the "2-substitution error-detection" property.

property for the channel γ such that $(u, v) \in \gamma$ iff u and v have equal lengths and differ in at most two positions.

Remark 5. *Let* $\mathbf{M} = \Sigma^*$. *Deciding whether a given transducer* $\hat{t}$ *is input-preserving is a known undecidable problem – see* [17]. *It turns out that deciding whether a given transducer* $\hat{t}$ *is input-altering is an undecidable problem as well. To see this we consider the complementary problem of whether* $\hat{t}$ *is* <u>not</u> *input-altering and we reduce the Post correspondence problem (PCP)* [37] *to that. In particular, for each instance* $T = \{(u_1, v_1), \ldots, (u_n, v_n)\}$ *of PCP, the single-state transducer with transitions* T *is not input-altering iff* T *is a YES instance of PCP.*

4.1. *A note on the conjunction of transducer properties*

As in the case of λ-NFAs, for any transducers $\hat{s}$ and $\hat{t}$, one can effectively construct a transducer realizing the relation $R(\hat{s}) \cup R(\hat{t})$. We denote this transducer as $(\hat{s} \cup \hat{t})$. Let τ be either of the type names 'al' and 'pr'. The closure under union implies that $\mathcal{P}^\tau_{\hat{s}} \cap \mathcal{P}^\tau_{\hat{t}} = \mathcal{P}^\tau_{\hat{s}\cup\hat{t}}$, that is, a language satisfies both $\mathcal{P}^\tau_{\hat{s}}$ and $\mathcal{P}^\tau_{\hat{t}}$ iff the language satisfies $\mathcal{P}^\tau_{\hat{s}\cup\hat{t}}$. To see this we simply note that the union of transducers preserves the type τ, and that the following holds for all languages L:

$$\hat{s}(L) \cap L = \emptyset \text{ and } \hat{t}(L) \cap L = \emptyset \iff (\hat{s} \cup \hat{t})(L) \cap L = \emptyset.$$

As an example, we have that *bifix* codes, codes that are both prefix and suffix, are definable in the transducer methodology. We note that these codes are not definable in the methodology of transitive and length-increasing binary relations [44].

4.2. *Some Theorems about Transducer Properties*

This subsection contains the main results about the previously defined properties. In the sequel, the size $|\hat{g}|$ of a finite state machine $\hat{g}$ is the sum of the numbers of states and transitions involved in the machine.

Lemma 6. *For every transducer $\hat{t}$, there is an input-preserving transducer $\hat{t}^{\mathrm{pr}}$ such that $\hat{t}^{\mathrm{pr}}(w) = \hat{t}(w) \cup w$, for all $w \in \Sigma^*$, and $\hat{t}^{\mathrm{pr}}$ can be constructed from $\hat{t}$ in linear time.*

Proof. We make a copy of $\hat{t}$ and a new state f such that f is final, in addition to those in $\hat{t}$. We also add a (λ/λ)-transition from the start state of $\hat{t}$ to f, and the transitions $(f, \sigma/\sigma, f)$ for all $\sigma \in \Sigma$. It follows that, for all words w, $\hat{t}^{\mathrm{pr}}(w) = \hat{t}(w) \cup w$, as required. □

Theorem 7. *The following statements hold true.*

(1) Every regular trajectory property is an input-altering transducer property, effectively.
(2) Every input-altering transducer property is an input-preserving transducer property, effectively.
(3) Every input-preserving transducer property is a 3-independence property.

Proof. The third statement follows from the definitions of 3-independence and input-preserving transducer properties. For the second statement, consider any input-altering transducer $\hat{t}$. Using Lemma 6, one verifies that, for any language L,

$$\hat{t}(L) \cap L = \emptyset \iff \forall\, u \in L,\ \hat{t}^{\mathrm{pr}}(u) \cap (L - u) = \emptyset. \tag{7}$$

This implies that, if $L \in \mathcal{P}^{\mathrm{al}}_{\hat{t}}$ then $L \in \mathcal{P}^{\mathrm{pr}}_{\hat{t}}$, as required.

For the first statement, we consider any regular expression $\bar{e}$ defining a trajectory property. We construct an input-altering transducer $\hat{t}$ such that, for any word u, $\hat{t}(u) = (u \amalg_{\bar{e}} \Sigma^+)$. This will imply that, for any language L, $(L \amalg_{\bar{e}} \Sigma^+) \cap L = \emptyset$ iff $\hat{t}(L) \cap L = \emptyset$, as required. For the transducer $\hat{t}$, we first construct an NFA $\hat{a}$ equivalent to $\bar{e}$. Consider a situation where $v \in (u \amalg_h \Sigma^+)$, for some trajectory $h \in L(\hat{a})$ and words u, v. The trajectory h is accepted in some path of $\hat{a}$. In this path, a transition of the form $(p, 0, q)$ indicates that a symbol from u is added in v, and a transition of the form $(p', 1, q')$ indicates that some symbol from Σ is added in v. It seems then that $\hat{t}$ should have the same states with $\hat{a}$, and transitions of the form $(p, \sigma/\sigma, q)$ and $(p', \lambda/\sigma, q')$, corresponding to the above transitions of $\hat{a}$. However, we have to make sure that at least one symbol outside of u gets added so that the transducer is input-altering. Thus, the construction is as follows. For each state q of $\hat{a}$, there are two states q^{yes} and q^{no} in $\hat{t}$, indicating whether or not the input has already been altered. The start state of $\hat{t}$ is s^{no}, where s is the start state of $\hat{a}$, and the final states of $\hat{t}$ are all states f^{yes}, where f is any final state of $\hat{a}$. For each transition, $(p, 0, q)$ of $\hat{a}$, we add the transitions $(p^{\mathrm{no}}, \sigma/\sigma, q^{\mathrm{no}})$ and

$(p^{\text{yes}}, \sigma/\sigma, q^{\text{yes}})$ in $\hat{t}$. Finally, for each transition $(p, 1, q)$ in $\hat{a}$, we add the transitions $(p^{\text{no}}, \lambda/\sigma, q^{\text{yes}})$ and $(p^{\text{yes}}, \lambda/\sigma, q^{\text{yes}})$ in $\hat{t}$. One verifies that, for every accepting path of $\hat{t}$ having (u, v) as a label, there is a "corresponding" trajectory h accepted by $\hat{a}$ such that $v \in (u \amalg_h \Sigma^+)$, and vice versa. □

Theorem 8. *The thin language property is definable in the input-altering transducer methodology, but it is not definable in the trajectory methodology.*

Proof. Fig. 2 shows that indeed the thin property is an input-altering transducer property. On the other hand, there is no regular expression describing the thin language property. Indeed, assume that $\bar{e}$ is an expression that does define thinness. Then, as $L = \{aa, bb\}$ is not a thin language, we have $(L \amalg_{\bar{e}} \Sigma^+) \cap L \neq \emptyset$. This implies that there is a trajectory $h \in L(\bar{e})$ and a nonempty word w such that $(aa \amalg_h w) \in L$, or $(bb \amalg_h w) \in L$. Then, $|h| = 2 + |w| > 2$, which is impossible as $|h|$ should also be the length of the L-word returned by the shuffle operation. We note that the same argument applies even if we assume that an arbitrary trajectory set (possibly non-regular) can be used to define thin languages. □

Theorem 9. *The following statements hold true.*

(1) The satisfaction problem for input-preserving transducer properties is decidable in polynomial time, assuming languages are described via NFAs.
(2) The satisfaction problem for input-altering transducer properties is decidable in time $O(|\hat{t}| \cdot |\hat{a}|^2)$, where $\hat{t}$ and $\hat{a}$ are the given transducer and NFA, respectively.
(3) The maximality problem is decidable for both input-preserving and -altering transducer properties, assuming languages are described via NFAs.
(4) For $\mathbf{M} = \Sigma^*$, the maximality problem for either of input-altering and -preserving transducer properties is PSPACE-hard.

Proof. For the first statement, we use results from [26]: For any transducer $\hat{t}$ and λ-NFA $\hat{b}$, there are, effectively in polynomial time, a transducer $\hat{t} \downarrow \hat{b}$ realizing $R(\hat{t}) \cap (L(\hat{b}) \times \Sigma^*)$ and a transducer $\hat{t} \uparrow \hat{b}$ realizing $R(\hat{t}) \cap (\Sigma^* \times L(\hat{b}))$. For the satisfaction problem, let $\hat{t}$ and $\hat{a}$ be the given transducer and NFA, and let $L = L(\hat{a})$. As $\hat{t}$ is input-preserving, (6) is equivalent to $\forall\, u \in L$, $|\hat{t}(u) \cap L| \leq 1$, which in turn is equivalent to whether the transducer $(\hat{t} \downarrow \hat{a}) \uparrow \hat{a}$ is functional. Thus (6) is decidable, as transducer functionality is polynomially decidable [13] – see also [14], [2].

For the second statement, assume that $\hat{t}$ is in standard form, and let $L = L(\hat{a})$. Deciding $\hat{t}(L) \cap L = \emptyset$ can be done as follows: First, use a product construction between $\hat{t}$ and $\hat{a}$ to get an automaton $\hat{b}$ accepting $\hat{t}(L)$ [42]. Then, use again a product construction between $\hat{a}$ and $\hat{b}$ to get an automaton accepting $\hat{t}(L) \cap L$, and test whether there is any accepting path in that automaton.

For the third statement, first consider any given input-preserving transducer $\hat{t}$ and NFA $\hat{a}$, and let $L = L(\hat{a})$. Assume that $L \in \mathcal{P}_{\hat{t}}^{\text{pr}}$. We show that L is *not*

$\mathcal{P}^{\mathrm{pr}}_{\hat{t}}$-maximal iff

$$\mathbf{M} \cap (\hat{t}(L) \cup \hat{t}^{-1}(L) \cup L)^{\mathrm{c}} \neq \emptyset. \tag{8}$$

The language on the left-hand side of (8) is an effectively regular language which can be constructed from $\hat{m}, \hat{a}$ and $\hat{t}$, and therefore (8) is decidable. Now we have that L is not $\mathcal{P}^{\mathrm{pr}}_{\hat{t}}$-maximal, iff there is a word $w \in \mathbf{M} - L$ such that $\forall\, u \in (L \cup w)$, $\hat{t}(u) \cap ((L \cup w) - u) = \emptyset$. One verifies that

$\forall\, u \in (L \cup w)$, $\hat{t}(u) \cap ((L \cup w) - u) = \emptyset$, iff

$\forall\, u \in L, \hat{t}(u) \cap ((L \cup w) - u) = \emptyset$, and $\hat{t}(w) \cap ((L \cup w) - w) = \emptyset$, iff

$\forall\, u \in L, \hat{t}(u) \cap ((L - u) \cup w) = \emptyset$, and $\hat{t}(w) \cap L = \emptyset$, iff

$\forall\, u \in L, \hat{t}(u) \cap (L - u) = \emptyset$, and $\forall\, u \in L, \hat{t}(u) \cap w = \emptyset$, and $\hat{t}(w) \cap L = \emptyset$, iff

$\hat{t}(L) \cap w = \emptyset$ and $\hat{t}(w) \cap L = \emptyset$, iff $w \notin (\hat{t}(L) \cup \hat{t}^{-1}(L))$,

where we have used the facts $\hat{t}(u) \cap (L - u) = \emptyset$, as L is in $\mathcal{P}^{\mathrm{pr}}_{\hat{t}}$, and $\hat{t}(w) \cap L = \emptyset \longleftrightarrow w \cap \hat{t}^{-1}(L) = \emptyset$. Statement (8) follows from this derivation.

For the case where $\hat{t}$ is input-altering, the maximality problem can be decided by first constructing the input-preserving transducer $\hat{t}^{\mathrm{pr}}$ – see Lemma 6 – and then deciding whether the given language L is $\mathcal{P}^{\mathrm{pr}}_{\hat{t}^{\mathrm{pr}}}$-maximal. For the correctness of this, one verifies that L is not $\mathcal{P}^{\mathrm{al}}_{\hat{t}}$-maximal iff there is $w \in \mathbf{M} - L$ with $\forall\, u \in (L \cup w)$: $\hat{t}(u) \cap (L \cup w) = \emptyset$, iff there is $w \in \mathbf{M} - L$ with $\forall\, u \in (L \cup w) : \hat{t}^{\mathrm{pr}}(u) \cap ((L \cup w) - u) = \emptyset$.

For the fourth statement, it is sufficient to show that deciding (8), for $\hat{t}$ input-preserving, is PSPACE-hard. First, recall that UNIVERSALREX is PSPACE-complete [15]. This is the problem of deciding whether $L(\bar{e}) = \Sigma^*$, for a given regular expression $\bar{e}$. Now UNIVERSALREX is reducible to our maximality problem as follows: Convert any given $\bar{e}$ to an equivalent NFA $\hat{a}$ – hence, $L = L(\hat{a}) = L(\bar{e})$. Construct a transducer $\hat{t}$ such that $\hat{t}(w) = w$, for all words w. Then also $\hat{t}^{-1}(w) = w$, for all words w. Obviously, $\hat{t}$ is input-preserving, and $L(\bar{e}) = \Sigma^*$, iff (8) is false, iff $L(\hat{a})$ is $\mathcal{P}^{\mathrm{pr}}_{\hat{t}}$-maximal. As before, the input-altering case can be handled via Lemma 6. □

As an application of the above theorems we obtain the following decidability results which, to our knowledge, are new.

Corollary 10. *The following problems are decidable.*

INPUT*: An NFA $\hat{a}$;* RETURN*: Is $L(\hat{a})$ thin-maximal?*

INPUT*: A channel transducer $\hat{t}$ and NFA $\hat{a}$;* RETURN*: Is $L(\hat{a})$ maximal error-detecting for $R(\hat{t})$?*

INPUT*: A channel transducer $\hat{t}$ and NFA $\hat{a}$;* RETURN*: Is $L(\hat{a})$ maximal error-correcting for $R(\hat{t})$?*

Proof. The first statement follows from the fact that thinness is an input-altering transducer property. Now recall that a channel is an input-preserving relation, so a channel transducer is simply an input-preserving transducer. The error-detection

case follows from the previous theorem when we observe that the "error-detection for $R(\hat{t})$" property is the input-preserving transducer property $\mathcal{P}^{\mathrm{pr}}_{\hat{t}}$, i.e., the property obtained using exactly $\hat{t}$ as the defining transducer. The error-correction case follows from the fact that a language is error-correcting for a channel γ iff it is error-detecting for $\gamma^{-1} \circ \gamma$, [27], and the fact that transducers are effectively closed under inverse and composition [3, 42]. □

5. More on the complexity of the maximality problem

The hardness results for the property maximality problem concern the general case where languages are described via NFAs and the property is part of the input. Here we show that the cause of the hardness is nondeterminism: even if we fix the property and restrict our attention to acyclic NFAs, i.e., those accepting finite languages, many of the problems remain hard. In particular, the hardness includes the "classical" code properties of unique decodability, prefix, suffix, bifix, and infix.

Theorem 11. *Let $\mathbf{M} = \Sigma^* = \{a, b\}^*$, and let τ be any of the code property names "uniquely decodable", "prefix", "suffix", "bifix", "infix", "hypercode". The problem of deciding whether a finite τ code, given via an acyclic NFA, is a maximal τ code is coNP-hard.*

Proof. Let us call the decision problem in question τ-FinNfaMax, and let FullFinNfa be the problem of deciding whether a given acyclic NFA accepts Σ^n, for some n. One verifies that the statement is an immediate logical consequence of the following two claims.

Claim 1 FullFinNfa is coNP-complete.

Claim 2 FullFinNfa is polynomially reducible to τ-FinNfaMax.

The first claim follows from the proof of the following fact on page 329 of [32]: Deciding whether two given star-free regular expressions are inequivalent is an NP-complete problem. Indeed, in that proof the first regular expression can be arbitrary, but the second regular expression represents the language Σ^n, for some positive integer n. Moreover, converting a star-free regular expression to an acyclic NFA is a polynomial time problem.

For the second claim, consider any acyclic NFA $\hat{a}$ and assume that it is already trim. To decide whether $L(\hat{a}) = \Sigma^n$, for some n, we first make sure that all words in $L(\hat{a})$ are of the same length by simply testing (using e.g. a breadth first search algorithm) whether all paths from the start state to any final state of $\hat{a}$ have equal lengths. If yes, we have that $L(\hat{a}) \subseteq \Sigma^n$, for some n, and that already $L(\hat{a})$ is a τ code (for all values of τ). Then we continue by deciding whether $\hat{a}$ is in τ-FinNfaMax. This decision settles our problem because of the following.

Claim 3 A τ code K with words of some fixed length n is maximal iff $K = \Sigma^n$.

The claim can be verified for each case of τ. For example, let $\tau =$ "prefix". For the 'if' part, as $K = \Sigma^n$, any word w outside of K must be a prefix of some word

in K (if $|w| < n$), or must contain a word of K as a prefix (if $|w| > n$). Hence, no new word can be added in K. The other cases can be handled analogously – for the "uniquely decodable" case, see page 41 of [4]. The 'only if' part follows from the 'if' part and the definition of maximality: K and Σ^n are maximal, and $K \subseteq \Sigma^n$; hence, $K = \Sigma^n$. □

We discuss maximality questions a little more in the last section of this paper. Next, we close this section with the observation that even for thin languages, testing maximality via NFAs is hard.

Remark 12. *Let $\Sigma = \{a\}$ and $\mathbf{M} = a^*$. The problem of deciding whether a language L (given via an NFA) is maximal is coNP-complete. To see this, first note that, as $\Sigma = \{a\}$, L is always thin, and L is thin-maximal iff $L = a^*$. In* [41]*, it is shown that deciding whether the language represented by a given regular expression $\bar{e}$ is* not *equal to a^* is an NP-complete problem. We also note that converting a regular expression to an NFA can be done in polynomial time.*

6. LAnguage SERver

Here we give a brief description of the architecture, current functionality and user interface of a web server for deciding the satisfaction problem about given languages and code properties. We call it LaSer [30], an acronym for Language Server. As customary with these applications, this is an ongoing project. However, at this point, the server is capable of performing the following tasks.

Task 1 Given NFA $\hat{a}$ and input-altering transducer $\hat{t}$, decide whether $L(\hat{a})$ satisfies $\mathcal{P}^{\mathrm{al}}_{\hat{t}}$.

Task 2 Given NFA $\hat{a}$ and NFA $\hat{b}$ over $\{0, 1\}$, decide whether $L(\hat{a})$ satisfies the regular trajectory property described by $\hat{b}$ – in fact by any regular expression equivalent to $\hat{b}$.

Task 3 If the answer to either of the above problems is negative, LaSer also returns two $L(\hat{a})$-words violating the desired property.

Although our presentation of the regular trajectory properties involves regular expressions, the current version of the server accepts the trajectory set via an NFA (Task 2). At this point a user would have to use another software (or set of libraries) to convert regular expressions to NFAs – one can use Grail [12] or the freely available FAdo set of libraries [10]. To our knowledge, there is no server of this type available on the internet – in particular with the capability to accept as input an unknown code property. In any case, we invite readers and users to test the server and send us their comments. We note that in [1], a C++ program is presented that is capable of generating DNA languages of a given cardinality satisfying various combinations of properties from a certain *fixed* set of properties. One of our next goals is to incorporate in our server a similar capability.

The user interface is very simple – see Fig. 5. The user provides the files contain-

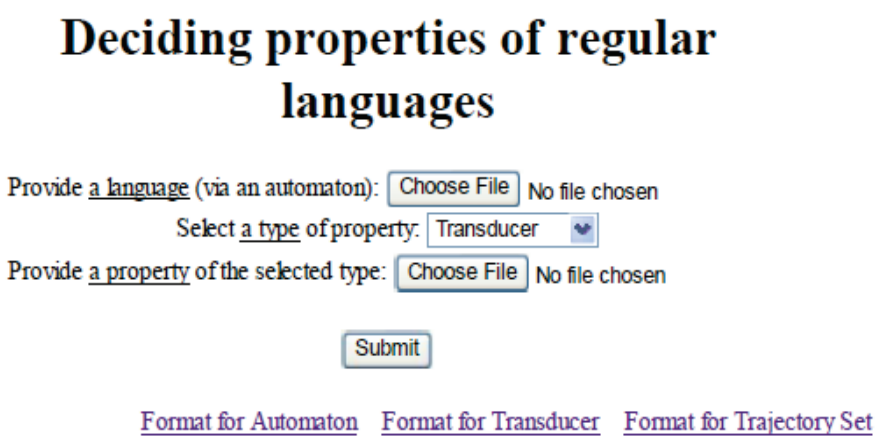

Fig. 5. The main user interface of LaSer

ing the NFA and transducer (Task 1), or the two NFAs (Task 2), and presses the submit button. In case the answer is NO, LaSer returns two words of the language violating the property (Task 3). The format for NFAs is the Grail format [12]. Here is an example of a Grail NFA accepting the language ab^*:

```
(START) |- 1
1 a 2
2 b 2
2 -| (FINAL)
```

For transducers, we assume that they are always in standard form. We use a Grail-like format, where we write `lambda` for the empty word. Here is an example of the Grail-like form of the transducer shown in Fig. 1:

```
(START) |- 0
0 a lambda 1
0 b lambda 1
1 a lambda 1
1 b lambda 1
1 a a 2
1 b b 2
2 a a 2
2 b b 2
1 -| (FINAL)
2 -| (FINAL)
```

LaSer consists of two layers, a web based user interface (UI) and an implementation of algorithms. UI was built using Django [6], a web framework for the Python programming language. The algorithms were implemented in the C++ programming language. We used features available in BOOST [5] (a set of open source C++ libraries) to integrate the implementation of algorithms with UI. It allowed us to take advantage of the execution speed of C++ and, at the same time, the convenience of Python and Django. Our application is hosted on a virtual machine with Ubuntu 8.10 and Apache server.

A limit: LaSer decides instances of the satisfaction problem using an incremental product construction of an NFA accepting the language $\hat{t}(L(\hat{a})) \cap L(\hat{a})$ – see Theorem 9. As the size of this NFA could grow very large, we have currently set 500,000 as the largest number of transitions that can be processed. So if more than this number of transitions need to be processed, LaSer simply interrupts the computation and returns a message.

7. Concluding remarks

We have presented a formal method for defining code properties, which fits well between the broad independence method and the formal regular trajectory method. We have discussed the decision problems of property satisfaction and maximality in the transducer methods, and presented some aspects of the related language server. We hope that our contribution will help bridging the gap between mathematical methods and implemented systems, a task that is often neglected.

We believe that the research direction of formal methods for language properties is a fruitful area with many interesting problems. We discuss this point in this and the next paragraphs. First, whether the property satisfaction and maximality problems can be answered for properties not definable in existing formal methods. Moreover, the problem of computing languages satisfying a given property is a natural next step – see [44, 1] for related works. The research on formal methods could lead to new and possibly better language servers.

Regarding the maximality problem, our hardness results concern the case where languages are described via NFAs. However, for fixed code properties and different descriptions for languages the problem can be of polynomial complexity. For example, in [24] it is shown that the problem can be decided in linear time for prefix codes described via DFAs, and in quadratic time for finite infix codes described via a list of words. At this point, we are not aware of the complexity of deciding the maximality of the suffix or infix property when the languages are described via DFAs, so we propose this as a question for future research.

Recall that the maximality problem requires testing the emptiness of the complement of the language $\hat{t}(L) \cup \hat{t}^{-1}(L) \cup L$, which results from the given L by applying transductions. Even when L is described via a DFA, applying transductions normally results in NFAs for which complementation is, in general, an exponential operation, and this appears to be the source of hardness for the maximality prob-

lem. A possible line of research then is to study the state complexity of the basic transduction operations that are needed in defining various code properties.

In analogy to a question of [7] about regular trajectory expressions, it is natural to ask whether it is decidable, for given transducers $\hat{t}$ and $\hat{s}$, if $\mathcal{P}^{\mathrm{al}}_{\hat{t}} = \mathcal{P}^{\mathrm{al}}_{\hat{s}}$ (or for what types of transducers this question is decidable). Furthermore, regarding code properties defined via multiple regular trajectory sets [8], it is known that already the property satisfaction problem is undecidable, which implies that that method is too general for the objectives set in this paper. However, it seems interesting to investigate types of multiple regular trajectory sets defining transducer properties.

Finally, it would be interesting to investigate how the property maximality problem can be decided in the formal method of implicational conditions [18].

References

[1] D. Bärmann. *Aufzählen von DNA-codes*. Diplomarbeit, Universität Potsdam, 2006, (in German).
[2] M.P. Béal, O. Carton, C. Prieur and J. Sakarovitch. Squaring transducers: An efficient procedure for deciding functionality and sequentiality. *Theoret. Computer Science* **292**:1 (2003), 45–63.
[3] J. Berstel. *Transductions and Context-Free Languages*. B. G. Teubner, Stuttgart, 1979..
[4] J. Berstel and D. Perrin. *Theory of Codes*. Academic Press, Orlando, 1985.
[5] Boost. http://www.boost.org/ Accessed on Oct. 18, 2010
[6] Django. http://www.djangoproject.com/ Accessed on Oct. 18, 2010
[7] M. Domaratzki. Trajectory-based codes. *Acta Informatica* **40**:6-7 (2004), 491–527.
[8] M. Domaratzki and K. Salomaa. Codes defined by multiple sets of trajectories. *Theoret. Computer Science* **366**:3 (2006), 182–193.
[9] K. Dudzinski and S. Konstantinidis. *Formal descriptions of code properties: decidability, complexity, implementation*. Technical report 2010-03, Math. and Comp. Sci., Saint Mary's University, 2010, pp 16.
[10] FAdo. http://www.ncc.up.pt/FAdo/fadoWWW.html Accessed on Oct. 18, 2010
[11] H. Fernau, K. Reinhardt and L. Staiger. Decidability of code properties. *RAIRO-Theor. Inf. Appl.* **41**:3 (2007), 243–259.
[12] Grail+. http://www.csd.uwo.ca/Research/grail/ Accessed on Oct. 18, 2010
[13] E.M. Gurari and O.H. Ibarra. A Note on Finite-Valued and Finitely Ambiguous Transducers. *Math. Systems Theory* **16** (1983), 61–66.
[14] T. Head and A. Weber. Deciding code related properties by means of finite transducers. In R. Capocelli, A. de Santis and U. Vaccaro (eds). *Sequences II, Methods in Communication, Security, and Computer Science*, 260–272. Springer Berlin, 1993.
[15] J.E. Hopcroft, R. Motwani and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd ed., 2001.
[16] S. Hussini, L. Kari and S. Konstantinidis. Coding Properties of DNA Languages. *Theoretical Computer Science* **290** (2003), 1557–1579.
[17] H. Johnson. Rational equivalence relations. *Theoret. Computer Science* **47**:1 (1986), 39–60.
[18] H. Jürgensen. Syntactic monoids of codes. *Acta Cybernetica* **14** (1999), 117–133.
[19] H. Jürgensen and S. Konstantinidis. The hierarchy of codes. In Z. Ésik (ed). *Fundamentals of Computation Theory, FCT'93. Lecture Notes in Computer Science* ***710***, 50–68. Springer-Verlag Berlin, 1993.

[20] H. Jürgensen and S. Konstantinidis. Codes. In [37], pp 511–607.
[21] H. Jürgensen, K. Salomaa and S. Yu. Transducers and the decidability of independence in free monoids. *Theoretical Computer Science* **134** (1994), 107–117.
[22] H. Jürgensen and S.S. Yu. Relations on free monoids, their independent sets, and codes. *Intern. J. Computer Mathematics* **40** (1991), 17–46.
[23] H. Jürgensen and S.S. Yu. *Dependence systems and hierarchies of families of languages*. Unpublished manuscript, 1995
[24] L. Kari and S. Konstantinidis. Language equations, maximality and error-detection. *J. Computer and System Sciences* **70** (2005), 157–178.
[25] L. Kari, S. Konstantinidis and P. Sosík. On properties of bond-free DNA languages. *Theoret. Computer Science* **334**:1-3 (2005), 131–159.
[26] S. Konstantinidis. Transducers and the properties of error-detection, error-correction, and finite-delay decodability. *J. Universal Computer Science* **8**:2 (2002), 278–291.
[27] S. Konstantinidis and P. Silva. Maximal error-detecting capabilities of formal languages. *J. Automata, Languages and Combinatorics* **13**:1 (2008), 55–71.
[28] N.H. Lâm. *Finite maximal infix codes*. Technical report 98/A2, Institute of Mathematics, Vietnam National Centre for Natural Science and Technology, 1998.
[29] N.H. Lâm. Finite maximal solid codes. *Theoretical Computer Science* **262**:1 (2001), 333–347.
[30] LaSer. http://laser.cs.smu.ca/transducer/ Accessed on Oct. 18, 2010
[31] V. Levenshtein. Certain properties of code systems. *Soviet Physics Dokl.* **6** (1962), 858–860.
[32] H. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation, 2nd ed.* Prentice Hall, 1998.
[33] A.A. Markov. Nonrecurrent Coding. *Problemy Kibernet.* **8** (1962), 169–180 (in Russian).
[34] A. Mateescu, G. Rozenberg and A. Salomaa. Shuffle on trajectories: Syntactic constraints. *Theoretical Computer Science* **197**:1-2 (1998), 1–56.
[35] A. Mateescu and A. Salomaa. Formal languages: An introduction and a synopsis. In [37], pp 1–39.
[36] G. Păun and A. Salomaa. Thin and slender languages. *Discrete Applied Mathematics* **61** (1995), 257–270.
[37] G. Rozenberg and A. Salomaa (eds). *Handbook of Formal Languages, Vol. 1*. Springer-Verlag, Berlin, 1997.
[38] A.A. Sardinas and G.W. Patterson. A necessary and suffcient condition for the unique decomposition of coded messages. *IRE Intern. Convention Record* **8** (1953), 104–108.
[39] H.J. Shyr. *Free Monoids and Languages*. Hon Min Book Company, Taichung, Taiwan, 1991, 2nd edition.
[40] H.J. Shyr and G. Thierrin. Codes and binary relations. *Séminaire d'Algèbre Paul Dubreil, Paris 1975–1976 (29ème Année), Lecture Notes in Mathematics* **586** (1977), 180–188.
[41] L.J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time (Preliminary report). *Proceedings of the 5th annual ACM symposium on Theory of computing* (1973), 1–9.
[42] S. Yu. Regular Languages. In [37], pp 41–110.
[43] S.S. Yu. *Languages and Codes*. Tsang Hai Publishing Co., Taichung, Taiwan, 2005.
[44] D.L. Van, K.V. Hung and P.T. Huy. Codes and length-increasing transitive binary relations. *Theoretical Aspects of Computing – ICTAC 2005, Lecture Notes in Computer Science* **3722** (2005), 29–48.