International Journal of Foundations of Computer Science
Vol. 22, No. 5 (2011) 1197–1209

DOI: 10.1142/S0129054111008647

World Scientific
www.worldscientific.com

OVERLAP-FREE LANGUAGES AND SOLID CODES

YO-SUB HAN*

Department of Computer Science, Yonsei University
Seoul 120-794, Republic of Korea
emmous@cs.yonsei.ac.kr

KAI SALOMAA

School of Computing, Queen's University
Kingston, Ontario K7L 3N6, Canada
ksalomaa@cs.queensu.ca

Received 10 January 2010
Accepted 3 March 2011
Communicated by Sheng Yu

Solid codes have a nice property called *synchronization property*, which is useful in data transmission. The property is derived from infix-freeness and overlap-freeness of solid codes. Since a code is a language, we look at solid codes from formal language viewpoint. In particular, we study regular solid codes (that are solid codes and regular). We first tackle the solid code decidability problem for regular languages and propose a polynomial time algorithm. We, then, investigate the decidability of the overlap-freeness property and show that it is decidable for regular languages but is undecidable for context-free languages. Then, we study the prime solid code decomposition of regular solid codes and propose an efficient algorithm for the prime solid code decomposition problem. We also demonstrate that a solid code does not always have a unique prime solid code decomposition.

Keywords: Solid codes; overlap-freeness; primality; regular languages.

1. Introduction

People use codes in various areas such as information processing, data compression, cryptography and information transmission [1, 12]. We can categorize codes by their defining properties (for example, *prefix-freeness*, *suffix-freeness* or *infix-freeness*) and use each family of codes in different applications [8, 9, 10]. Since codes are sets of strings, they are closely related to formal language theory; a code is a *language*. These code conditions define a proper subfamily of a given language family. For instance, prefix-freeness defines a family of prefix-free regular languages and it is a proper subfamily of regular languages [1].

*Corresponding author.

We study solid codes: a set S of strings is a *solid code* [19] or a *code without overlaps* [15] if S satisfies the following conditions:

(1) no string of S is a substring of another string of S (infix-freeness) and
(2) no prefix of a string in S is a suffix of a string in S (overlap-freeness).

In other words, S has to be an infix code and all strings of S should not overlap. Solid codes have the synchronization property, which is useful in data transmission [12]. Solid codes lead us to investigate infix codes and overlap-freeness in regular languages. As a continuation of our research of subfamilies of regular languages, it is natural to examine overlap-free regular languages and regular solid codes. Note that we have already studied infix-free regular languages [5].

We define some basic notions in Section 2. In Section 3, we design efficient algorithms that determine whether or not a given regular language is overlap-free or a solid code in polynomial time. We also prove that overlap-freeness is undecidable for context-free languages. Then, in Section 4, we examine the solid code primality and prime decomposition. We develop an algorithm for computing a prime solid code decomposition for a regular solid code when it is not prime. We also demonstrate that a solid code does not have a unique prime solid code decomposition.

2. Preliminaries

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ. The size $|\Sigma|$ of Σ is the number of characters in Σ. A language over Σ is any subset of Σ^*. The symbol $\emptyset$ denotes the empty language and the symbol λ denotes the null string. For strings x, y and z, we say that x is a *prefix* of y if $y = xz$. Similarly, x is a *suffix* of y if $y = zx$. We define x to be an *infix* (or substring) of y if $y = uxv$ for two strings u, v. We define a (regular) language L to be prefix-free if for any two distinct strings x and y in L, x is not a prefix of y. Similarly, we can define suffix-free and infix-free languages. Given a string x in a set X of strings, let x^R be the reversal of x, in which case $X^R = \{x^R \mid x \in X\}$.

An FA A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. If F consists of a single state f, then we use f instead of $\{f\}$ for simplicity. Let $|Q|$ be the number of states in Q and $|\delta|$ be the number of transitions in δ. We define the size of A to be $|A| = |Q| + |\delta|$. For a transition $\delta(p, a) = q$ in A, we say that p has an *out-transition* and q has an *in-transition*. Furthermore, p is a *source state* of q and q is a *target state* of p. We say that A is *non-returning* if the start state of A does not have any in-transitions and A is *non-exiting* if all final states of A have no out-transitions.

A string x over Σ is accepted by A if there is a labeled path from s to a final state such that this path reads x. We call this path an *accepting path*. Then, the language $L(A)$ of A is the set of all strings spelled out by accepting paths in A. We say that a state of A is *useful* if it appears in an accepting path in A; otherwise, it

is *useless*. Unless otherwise mentioned, in the following we assume that all states are useful.

For complete background knowledge in automata theory, the reader may refer to textbooks [7, 20].

Definition 1. *Let L be a language and u, v, w be strings over Σ.*

(1) L is infix-free *if $v, uvw \in L$ implies that $u = w = \lambda$.*
(2) L is overlap-free *if $uv, vw \in L$ implies that one of u, v, w has to be λ.*
(3) L is a solid code *if L is infix-free and overlap-free.*

Note that every infix-free language is a code. However, an overlap-free language may not be a code. For example, $L = \{a, ab, aab\}$ is overlap-free but is not a code since $(a)(ab) = (aab)$. On the other hand, note that for any $w \in \Sigma^+$, the singleton set $\{w\}$ is a code, however, it need not be overlap-free, for example, when $w = aba$. Strings w such that $\{w\}$ is overlap-free are called *unbordered* [12].

3. Decidability of Overlap-Free Languages and Solid Codes

Although an overlap-free language may not be a code, overlaps are important to define solid codes. Moreover, overlaps are often used in various proofs in coding theory [12]. Overlaps are also crucial for decoding and synchronization [11, 12]. We first study overlap-free languages and then, look at solid codes.

3.1. *Overlap-free languages*

We examine the derivative operation [2] since overlaps naturally lead to the operation. The *derivative* $x\backslash L$ of a language L with respect to a string x is the language $\{y \mid xy \in L\}$.

Theorem 2. *Given a language L and a string $x \neq \lambda$, let*

$$L_x = ((x\backslash L) - \{\lambda\}) \cdot \Sigma^+.$$

L is overlap-free if and only if $L_x \cap L = \emptyset$ for any string x.

Proof.
$\Longrightarrow$ Assume that $L_x \cap L \neq \emptyset$ for a string x. Choose $w \in L_x \cap L$ and write $w = w_1 w_2$, where $w_1 \in (x\backslash L) - \{\lambda\}$ and $w_2 \in \Sigma^+$. By the definition of the derivative operation, it follows that $xw_1 \in L$ and this together with $w_1 w_2 \in L$ implies that L is not overlap-free.
$\Longleftarrow$ Assume that L is not overlap-free. This implies that there are strings u, v, w such that $uv, vw \in L$ and $u, v, w \neq \lambda$. Note that

$$v \in (u\backslash L) - \{\lambda\},$$

$$vw \in ((u\backslash L) - \{\lambda\}) \cdot \Sigma^+ = L_u.$$

Then $vw \in L_u \cap L$—a contradiction.

Therefore L is overlap-free if and only if $L_x \cap L = \emptyset$. □

Assume that L is regular and given by an FA. Then, since regular languages are effectively closed under the derivative operation, Theorem 2 shows that it is decidable whether or not L is overlap-free. However, the algorithm in the proof of Theorem 2 requires lots of computation. Recently, Han et al. [5] introduced the state-pair graph for FAs and demonstrated that it is useful to determine certain structural properties that a code must have. We now show that we can decide whether or not L is overlap-free quickly when L is regular using state-pair graphs.

Given an FA $A = (Q, \Sigma, \delta, s, F)$ for L, we assign a unique number for each state in A from 1 to m, where $m = |Q|$.

Definition 3 (Han et al. [5]) *Given an FA $A = (Q, \Sigma, \delta, s, F)$, we define the state-pair graph $G_A = (V_G, E_G)$, where V_G is a set of nodes and E_G is a set of edges, as follows:*

$$V_G = \{(i, j) \mid i \text{ and } j \in Q\} \text{ and}$$
$$E_G = \{((i, j), a, (x, y)) \mid (i, a, x) \text{ and } (j, a, y) \in \delta \text{ and } a \in \Sigma\}.$$

We define the size $|G_A|$ of G_A to be the number of nodes plus the number of edges. Note that $|G_A| \leq |Q|^2 + |\delta|^2$.

The crucial property of state-pair graphs is that if there is a string w spelled out by two distinct paths in A, for example, one path is from i to x and the other path is from j to y, then, there is a path from (i, j) to (x, y) in G_A that spells out the same string w. Note that state-pair graphs do not require given FAs to be deterministic. The construction in Definition 3 assumes that A does not have null transitions, that is λ-transitions. However, it is easy to modify the state-pair graph construction that can handle λ-transitions by performing the λ-reachability test for computing E_G. For details on the λ-transition case, the readers may refer to Han et al. [5].

Theorem 4. *Given an FA $A = (Q, \Sigma, \delta, s, F)$, $L(A)$ is overlap-free if and only if the state-pair graph G_A for A has no path from $(1, i)$ to (j, k), where*

(1) i is a state that has an in-transition.
(2) j is a state that has an out-transition.
(3) k is a final state in F.
(4) 1 denotes the start state.

Proof.
$\Longrightarrow$ Assume that there is a path from $(1, i)$ to (j, k) that spells out a string w in G_A. This implies that there are two distinct paths in A, where one is from 1 to j and the other is from i to k and both paths spell out w. Since j has an out-transition and A has only useful states, there should be a path from j to a final state. Let z

be the string spelled out by this path from j. Then, we know that A accepts wz. Similarly, there should be a path from 1 to i since i should be reachable in A. Let y be the string spelled out by this path. (It may be possible that i is the start state.) Thus, A accepts yw. Note that

- z can be chosen to be non-empty because j has an out-transition and all state are useful,
- y can be chosen to be non-empty because i has an in-transition and all states are useful.

Therefore, A accepts both yw and wz, and w, y, z are not λ—a contradiction.
$\Longleftarrow$ Assume that $L(A)$ is not overlap-free. This implies that there are strings u, v, w such that $uv, vw \in L(A)$ and $u, v, w \neq \lambda$. Since $vw \in L(A)$, there is a corresponding path for vw in A. Let j be the state that we reach after reading v from the start state. This implies that j should have an out-transition with label w_1, which is the first character of w. Similarly, there must be an accepting path for uv in A. Let i be the state that we reach after reading u. Then, it is clear that i has an in-transition. Furthermore, we should reach to a final state k from i after reading v since uv is accepted by A. Therefore, there must be a path from $(1, i)$ to (j, k) that satisfies all four conditions above—a contradiction. □

We can check the existence of such a path of Theorem 4 in linear time in the size of G_A using Depth-First Search (DFS) [3]. Thus, we obtain the following result from Definition 3 and Theorem 4:

Theorem 5. *Given an FA $A = (Q, \Sigma, \delta, s, F)$, we can determine whether or not $L(A)$ is overlap-free in $O(|Q|^2 + |\delta|^2)$ worst-case time.*

Theorem 5 shows that given a regular language L, it is decidable whether or not L is overlap-free. Next, we investigate the decidability problem when L is not regular. Note that for most codes, (such as prefix, suffix or inter codes) it is undecidable when L is a linear language [12, 13]. We establish a similar result for overlap-freeness.

Theorem 6. *There is no algorithm that determines whether or not a given linear language L is overlap-free.*

Proof. Let Σ be an alphabet and let (U, V) be an instance of Post's Correspondence Problem [18], where

$$U = (u_0, u_1, \ldots, u_{n-1}) \text{ and } V = (v_0, v_1, \ldots, v_{n-1})$$

for a positive integer $n \in \mathbb{N}$ and $u_0, u_1, \ldots, u_{n-1}, v_0, v_1, \ldots, v_{n-1} \in \Sigma^*$. A solution to (U, V) is a pair (m, I), where m is a positive integer and I is an m-tuple of integers, $I = (i_0, i_1, \ldots, i_{m-1})$ such that

$$i_j \in \{0, 1, \ldots, n-1\} \text{ for } 0 \leq j \leq m-1 \text{ and } u_{i_0} u_{i_1} \cdots u_{i_{m-1}} = v_{i_0} v_{i_1} \cdots v_{i_{m-1}}.$$

Without loss of generality, we assume that the symbols 0, 1, #, \$ and ϕ are not in Σ. Let $\Sigma' = \Sigma \cup \{0, 1, \#, \$, \phi\}$. For any nonnegative integer i, let $\beta(i)$ be the shortest binary representation of i.

Consider a linear grammar $G = (N, \Sigma', P, S)$ for L, where

$N = \{S, T_U, T_V\}$ is the nonterminal alphabet,
Σ' is the terminal alphabet,
S is the sentence symbol and
the rules in P are

$$S \to \#\beta(i)\phi T_U u_i \mid \#\beta(i)\$u_i \mid \beta(i)\phi T_V v_i\% \mid \beta(i)\$v_i\%$$
$$T_U \to \beta(i)\phi T_U u_i \mid \beta(i)\$u_i$$
$$T_V \to \beta(i)\phi T_V v_i \mid \beta(i)\$v_i$$

for $i \in \{0, 1, \ldots, n-1\}$.

Namely, $L(G)$ is the language that consists of all the strings

$$\#\beta(i_{m-1})\phi\beta(i_{m-2})\phi \cdots \beta(i_0)\$u_{i_0} \cdots u_{i_{m-2}} u_{i_{m-1}} \text{ and}$$
$$\beta(i_{m-1})\phi\beta(i_{m-2})\phi \cdots \beta(i_0)\$v_{i_0} \cdots v_{i_{m-2}} v_{i_{m-1}}\%,$$

for all $m \in \mathbb{N}$ and $i_j \in \{0, 1, \ldots, n-1\}$ for $0 \leq j \leq m-1$.

Note that (U, V) has a solution if and only if $L(G)$ is not overlap-free. Therefore, it is undecidable whether or not $L(G)$ is overlap-free since Post's Correspondence Problem is undecidable [7, 18]. □

Theorem 6 shows that the overlap-freeness of a context-free language is undecidable. Before we move to solid codes, we study the closure properties of overlap-free languages. A subfamily of languages with certain code properties is often closed under catenation. For example, prefix-free languages, bifix-free languages, infix-free languages and outfix-free languages are all closed under catenation, respectively [1, 9].

Theorem 7. *The family of overlap-free (regular) languages is closed under intersection but not under catenation, union, complement or Kleene plus.*

Proof. Let L_1 and L_2 be overlap-free languages. Except for the intersection case, we prove results by giving examples.

- Let $L = L_1 \cap L_2$. Assume that L is not overlap-free. It implies that there are two strings $u, v \in L$ such that u and v overlap with each other. Since L is the intersection of L_1 and L_2, u and v must be L_1 and L_2 and, thus, L_1 and L_2 are not overlap-free—a contradiction. Therefore, overlap-free languages are closed under intersection.
- Let $L = L_1 \cdot L_2$. If $L_1 = L_2 = \{ab\}$, then $L = \{abab\}$ is not overlap-free. Therefore, overlap-free languages are not closed under catenation.
- Let $L = L_1 \cup L_2$. If $L_1 = \{ab\}$ and $L_2 = \{ba\}$ then $L = \{ab, ba\}$ is not overlap-free. This means that overlap-free languages are not closed under union.

Complement and Kleene plus cases can be proved straightforwardly. □

In fact, if L is overlap-free and L' is an arbitrary language, then $L \cap L'$ is always overlap-free. We also establish the following closure property under morphism and inverse morphism.

Theorem 8.

(a) *The family of overlap-free languages is closed under inverse non-erasing morphisms. The family of overlap-free languages is not closed under general inverse morphisms.*

(b) *The family of overlap-free languages is not closed under morphisms or non-erasing morphisms.*

Proof.

(a) Consider an overlap-free language $L \subseteq \Sigma^*$ and an arbitrary non-erasing morphism $\varphi : \Omega^* \to \Sigma^*$. For the sake of contradiction assume that $\varphi^{-1}(L)$ is not overlap-free. Thus, there exist $u, v, w \in \Omega^+$ such that $uv, vw \in \varphi^{-1}(L)$. This means that $\varphi(uv) = \varphi(u)\varphi(v) \in L$ and $\varphi(vw) = \varphi(v)\varphi(w) \in L$. Since φ is non-erasing, $\varphi(v) \neq \lambda$ and this contradicts the fact the L is overlap-free.

To see that overlap-free languages are not closed under inverse general morphisms choose $\Sigma = \{a, b\}$, $L = \{a\}$ and consider the morphism $\phi : \Sigma^* \to \Sigma^*$ defined by $\phi(a) = a$, $\phi(b) = \lambda$. Now L is overlap-free but, for example, $ba, ab \in \phi^{-1}(L)$.

(b) The non-closure under direct morphisms is straightforward and is left as an exercise for the reader. □

Remark that the family of (regular) solid codes is closed under inverse non-erasing morphisms [14].

3.2. *Solid codes*

We say that a language L is a solid code if and only if L is overlap-free and an infix code. We design an algorithm that determines whether or not a given regular language is a solid code. Note that it is undecidable for linear languages [12] and it is decidable for regular languages [14]. Given a regular language L, we can check whether or not L is an infix code in quadratic time [5] and can check whether or not L is overlap-free in quadratic time as shown in Theorem 5, where both algorithms are based on state-pair graphs. Therefore, we know that we can determine it in polynomial time.

Theorem 9. *Given an FA $A = (Q, \Sigma, \delta, s, f)$, $L(A)$ is a solid code if and only if the state-pair graph G_A for A has*

(1) no path from $(1, i)$ *to* (m, k)*, apart from* $(1, 1)$ *to* (m, m)*, where* $1 \leq i \leq m$ *and* $1 \leq k \leq m$ *(This is the infix code condition.)*
(2) no path from $(1, i)$ *to* (j, m)*, where* $i \neq 1$ *and* $j \neq m$*. (This is the overlap-free condition.)*

Proof. Han et al. [5] proved the first condition is valid for an infix code. The second condition is more restrict than the condition in Theorem 5 since we do not need to consider infix codes. Note that $L = \{a, ab\}$ is not an infix code but is overlap-free and Theorem 5 has to consider this case as well. □

Theorem 9 gives an algorithm that determines whether or not $L(A)$ is a solid code in polynomial time.

Theorem 10. *Given an FA* $A = (Q, \Sigma, \delta, s, f)$*, we can determine whether or not* $L(A)$ *is a solid code in* $O(|Q|^2 + |\delta|^2)$ *worst-case time using its state-pair graph.*

Note that in both Theorems 9 and 10, we assume that there is a single final state. This is because all infix FAs must be non-exiting; namely, no final state has an out-transition. This implies that all final states are equivalent and, thus, can be merged into a single final state.

4. Prime Decomposition for Regular Solid Codes

Decomposition is the reverse operation of catenation. If $L = L_1 \cdot L_2$, then L is the catenation of L_1 and L_2 and $L_1 \cdot L_2$ is a decomposition of L. We call L_1 and L_2 *factors* of L. Note that every language L has a decomposition, $L = \{\lambda\} \cdot L$, where L is a factor of itself. We call $\{\lambda\}$ a *trivial* language. We define a language L to be *prime* if $L \neq L_1 \cdot L_2$, for any non-trivial languages L_1 and L_2. Then, a prime decomposition of L is a decomposition $L = L_1 L_2 \cdots L_k$, where each L_i, $1 \leq i \leq k$, is a prime language.

Mateescu et al. [16, 17] showed that the primality of regular languages is decidable and the prime decomposition of a regular language is not unique. They also conjectured that the primality test of a regular language is NP-complete. On the other hand, if we preserve certain properties for each factor language, then we can compute a prime decomposition in polynomial time. For example, Czyzowicz et al. [4] showed that for a given prefix-free regular language L, the prime prefix-free decomposition is unique and the decomposition can be computed in $O(m)$ worst-case time, where m is the size of the minimal DFA for L. Han et al. [5] investigated the prime infix-free decomposition of infix-free regular languages and demonstrated that the prime infix-free decomposition is not unique. On the other hand, the prime outfix-free decomposition of outfix-free regular languages is unique [6].

We say a language L is a regular solid code if L is regular and a solid code. Now we investigate prime regular solid codes and prime decomposition for regular solid codes.

4.1. *Prime Regular Solid Codes*

Definition 11. *We define a regular language L to be a* prime *solid code if $L \neq L_1 \cdot L_2$, for any regular solid codes L_1 and L_2.*

From now on, when we say prime, we mean prime regular solid code.

Definition 12. *We define a state b in a DFA A to be a* bridge candidate state *if the following conditions hold:*

(1) State b is neither a start nor a final state.
(2) For any string $w \in L(A)$, its path in A must pass through b.
(3) State b is not in any cycles in A.

Given a solid code DFA $A = (Q, \Sigma, \delta, s, f)$ with a bridge candidate state $b \in Q$, we can partition A into two subautomata A_1 and A_2 as follows: $A_1 = (Q_1, \Sigma, \delta_1, s, b)$ and $A_2 = (Q_2, \Sigma, \delta_2, b, f)$, where Q_1 is a set of states that appear on some path from s and b in A, δ_1 is a set of transitions that appear on some path from s and b in A, $Q_2 = Q - Q_1 \cup \{b\}$ and $\delta_2 = \delta - \delta_1$. Note that the second requirement in Definition 12 ensures that the decomposition of $L(A)$ is $L(A_1) \cdot L(A_2)$ and the third requirement is from the property that a solid code FA must be non-returning and non-exiting since it is infix-free. Now we are ready to define bridge states:

Definition 13. *We define a candidate bridge state $b \in Q$ to be a* bridge state *if, for the automata A_1 and A_2 constructed as above, $L(A_1)$ and $L(A_2)$ are solid codes.*

We show the relationship between bridge states and prime regular solid codes.

Theorem 14. *A regular solid code L is prime if and only if the minimal DFA A for L does not have any bridge states.*

Proof. Let s denote the start state and f denote the final state in A.
$\Longrightarrow$ Assume that A has a bridge state q. We partition A into two automata A_1 and A_2 at q. Now s is the start state and q is the final state of A_1 and q is the start state and f is the final state of A_2. Note that $L = L(A_1) \cdot L(A_2)$ and $L(A_1)$ and $L(A_2)$ are solid codes because of the bridge state properties—a contradiction.
$\Longleftarrow$ Assume that L is not prime. This implies that L can be decomposed into two regular solid codes L_1 and L_2. Let A_1 and A_2 be minimal DFAs for L_1 and L_2, respectively. Since A_1 and A_2 are non-returning and non-exiting, there is only one start state and one final state for each minimal DFA. We catenate A_1 and A_2 by merging the final state of A_1 and the start state of A_2 into a single state q. Then, the catenated automaton is the minimal DFA for $L(A_1) \cdot L(A_2)$, which is L, and it has a bridge state q—a contradiction.

Therefore, L is prime if and only if A has no bridge states. □

We determine whether or not a given regular solid code $L(A)$ is prime based on bridge states. Next, when $L(A)$ is not prime and, thus, A has bridge states, we show how to obtain a prime decomposition for $L(A)$ using these bridge states.

4.2. *Prime Decomposition for Regular Solid Codes*

By a prime solid code decomposition of a regular solid code L, we mean the representation of L as a product $L_1 \cdot \ldots \cdot L_k$, $k \geq 1$, where each L_i is a regular solid code, $i = 1, \ldots, k$. If L is prime, then L itself is a prime decomposition. Thus, given a regular solid code L, we, first, determine whether or not L is prime. If L is not prime, then there should be some bridge state(s) and we decompose L using the bridge state(s). Let A_1 and A_2 be two subautomata partitioned at a bridge state for L. If both $L(A_1)$ and $L(A_2)$ are prime, then a prime decomposition of L is $L(A_1) \cdot L(A_2)$. Otherwise, we repeat the preceding procedure for a non-prime solid code.

Let B denote a set of bridge states for a given minimal DFA A. The number of states in B is at most m, where m is the number of states in A. Note that once we partition A at $b \in B$ into A_1 and A_2, then only the states in $B - \{b\}$ can be bridge states in A_1 and A_2. Note that it is not necessary for all remaining states to be a bridge state. Therefore, we can determine the primality of $L(A)$ by checking whether or not A has bridge states. Moreover, we can compute a prime decomposition of $L(A)$ using these bridge states. Since there are at most m bridge states in A, we can compute a prime decomposition of $L(A)$ after a finite number of decompositions at bridge states.

We first compute all candidate bridge states and, then we determine whether or not each candidate bridge state is a bridge state.

Proposition 15 (Han et al. [5]) *Given a minimal DFA $A = (Q, \Sigma, \delta, s, f)$, we can identify all candidate bridge states in $O(|Q| + |\delta|)$ worst-case time.*

Let CBS denote a set of candidate bridge states that we compute from a solid code DFA A based on Proposition 15. Once we obtain CBS, for each state $b_i \in CBS$, we check whether or not two subautomata A_1 and A_2 partitioned at b_i are solid codes. If both A_1 and A_2 are solid code DFAs, then L is not prime and, thus, we decompose L into $L(A_1) \cdot L(A_2)$ and continue to check and decompose for each A_1 and A_2, respectively, using the remaining states in $CBS - \{b_i\}$.

Theorem 16. *Given a DFA $A = (Q, \Sigma, \delta, s, f)$ for a regular solid code, we can determine the primality of $L(A)$ in $O(m^3)$ worst-case time and compute a prime decomposition for $L(A)$ in $O(m^4)$ worst-case time, where $m = |Q|$.*

Proof. Since the size of CBS is at most m and it takes $O(m^2)$ time for each candidate state in CBS to determine whether or not $L(A_1)$ and $L(A_2)$ are solid codes as shown in Theorem 10, the total running time for determining primality of $L(A)$ is $O(m) \times O(m^2) = O(m^3)$ in the worst-case.

If a candidate state $b_i \in CBS$ turns out to be a bridge state, then we partition A into A_1 and A_2 at b_i and repeat the procedure for $L(A_1)$ and $L(A_2)$, respectively, using the remaining candidate states in $CBS - \{b_i\}$. We continue this partitioning until we cannot decompose anymore. Therefore, the total time complexity for computing a prime decomposition of $L(A)$ is $O(m^4)$ in the worst-case. □

The algorithm for computing a prime decomposition for $L(A)$ in Theorem 16 looks similar to the algorithm for the regular infix code case studied by Han et al. [5]. However, there is one big difference between these two algorithms because of the different closure properties of two families: In fact, Han et al. [5] speeded up their algorithm by linear factor based on the fact that infix codes are closed under catenation whereas solid codes are not [14].

We observe that a bridge state b_i of a minimal DFA A may not be a bridge state anymore if A is partitioned at a different bridge state b_j. It hints that the prime decomposition for a regular solid code may not be unique. Note that the prime prefix-free decomposition for a regular prefix code is unique [4] whereas the prime infix-free decomposition for a regular infix code is not unique [5]. Since solid codes are a proper subfamily of both prefix codes and infix codes, it is natural to examine the uniqueness of prime decomposition for regular solid codes.

Example 17. *The following example demonstrates the non-uniqueness of prime solid code decomposition.*

$$L(c(ab+ba)d) = \begin{cases} L_1(c(ab+ba)) \cdot L_2(d). \\ L_2(c) \cdot L_3((ab+ba)d). \end{cases}$$

The language L is a regular solid code but not prime and it has two different prime decompositions, where L_1, L_2 and L_3 are prime and regular solid codes.

5. Conclusions

Solid codes are a language that is overlap-free and infix-free. We notice that overlap-freeness is an interesting property since, not like similar properties (such as prefix-freeness or suffix-freeness), it does not define a code. Nevertheless, it is an important property used in several proofs in coding theory [12]. Moreover, overlap-freeness is crucial for defining solid codes. Thus, we have examined the family of overlap-free regular languages and regular solid codes.

We have proposed algorithms that determine whether or not a given regular language is overlap-free or a solid code using its FA. We have also shown that it is undecidable whether or not a given context-free language is overlap-free. We have examined prime regular solid codes and designed a polynomial time algorithm that computes a prime decomposition for a regular solid code. We have then demonstrated that a solid code does not have a unique prime solid code decomposition.

We have not considered the question of determining whether a given overlap-free regular language L is prime, that is, whether L can be written in a non-trivial way as a product of overlap-free languages. Note that the definition of primality (and the algorithm used to determine primality) of a solid code (or some of the other usual code classes) relies essentially on the fact that solid codes are closed under concatenation. Thus, to determine whether or not a regular solid code L is prime it is sufficient to determine whether there exist regular solid codes L_1 and L_2 such that

$L = L_1 \cdot L_2$. However, since overlap-free languages are not closed under concatenation (Theorem 7), it may be the case that a regular overlap-free language L has no decomposition $L_1 \cdot L_2$ where L_i is overlap-free, $i = 1, 2$, but L can be written as a product $L'_1 \cdots L'_k$ for some $k > 2$, where each L'_i, $i = 1, \ldots, k$, is overlap-free. For example, consider the overlap-free language $L_0 = \{bbc, bac, bbca, baca, bbcaaa, bacaaa\}$. We can write $L_0 = \{b\}\cdot\{a, b\}\cdot\{c, ca, caaa\}$ where each of the three factors is overlap-free. However, it is easy to verify that L_0 has three different decompositions into a product of two non-trivial factors and in each of these decompositions one of the languages contains overlaps.

If we use the more general definition of primality for overlap-free languages that prohibits decompositions into any finite number of overlap-free factors, then it is not clear even whether this property is decidable for regular overlap-free languages. Primality of overlap-free languages remains a topic for future research.

Acknowledgments

We wish to thank the referee for the careful reading of the paper and many valuable suggestions.

Han was supported by the IT R&D program of MKE/IITA 2008-S-024-01 and the Basic Science Research Program through NRF funded by MEST (2010-0009168). Salomaa was supported by the Natural Sciences and Engineering Research Council of Canada Grant OGP0147224.

References

[1] J. Berstel and D. Perrin. *Theory of Codes*. Academic Press, Inc., 1985.
[2] J. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11:481–494, 1964.
[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
[4] J. Czyzowicz, W. Fraczak, A. Pelc, and W. Rytter. Linear-time prime decomposition of regular prefix codes. *International Journal of Foundations of Computer Science*, 14:1019–1032, 2003.
[5] Y.-S. Han, Y. Wang, and D. Wood. Infix-free regular expressions and languages. *International Journal of Foundations of Computer Science*, 17(2):379–393, 2006.
[6] Y.-S. Han and D. Wood. Outfix-free regular languages and prime outfix-free decomposition. *Fundamenta Informaticae*, 81(4):441–457, 2007.
[7] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 2 edition, 1979.
[8] M. Ito, H. Jürgensen, H.-J. Shyr, and G. Thierrin. N-prefix-suffix languages. *International Journal of Computer Mathematics*, 30:37–56, 1989.
[9] M. Ito, H. Jürgensen, H.-J. Shyr, and G. Thierrin. Outfix and infix codes and related classes of languages. *Journal of Computer and System Sciences*, 43:484–508, 1991.
[10] H. Jürgensen. Infix codes. In *Proceedings of Hungarian Computer Science Conference*, 25–29, 1984.
[11] H. Jürgensen. Synchronization. *Information and Control*, 206(9-10):1033–1044, 2008.
[12] H. Jürgensen and S. Konstantinidis. Codes. In G. Rozenberg and A. Salomaa, editors,

Word, Language, Grammar, volume 1 of *Handbook of Formal Languages*, 511–607. Springer-Verlag, 1997.

[13] H. Jürgensen, K. Salomaa, and S. Yu. Decidability of the intercode property. *Elektronische Informationsverarbeitung und Kybernetik*, 29(6):375–380, 1993.

[14] H. Jürgensen and S. S. Yu. Solid codes. *Elektronische Informationsverarbeitung und Kybernetik*, 26(10):563–574, 1990.

[15] V. I. Levenshtein. Maximum number of words in codes without overlaps. *Problemy Peredachi Informatsii*, 6(4):88–90, 1970. In Russian. English translation in Problems Inform. Transmission.

[16] A. Mateescu, A. Salomaa, and S. Yu. On the decomposition of finite languages. Technical Report 222, TUCS, 1998.

[17] A. Mateescu, A. Salomaa, and S. Yu. Factorizations of languages and commutativity conditions. *Acta Cybernetica*, 15(3):339–351, 2002.

[18] E. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.

[19] H. Shyr and S. Yu. Intercodes and some related properties. *Soochow J. Math.*, 16(1):95–107, 1990.

[20] D. Wood. *Theory of Computation*. John Wiley & Sons, Inc., New York, NY, 1987.