Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Information Processing Letters 86 (2003) 247–253

Information Processing Letters

www.elsevier.com/locate/ipl

The hardest linear conjunctive language

Alexander Okhotin

School of Computing, Queen's University, Kingston, ON, Canada K7L3N6

Received 20 September 2002; received in revised form 17 December 2002

Communicated by L. Boasson

Abstract

This paper demonstrates that the P-complete language of yes-instances of Circuit Value Problem under a suitable encoding can be generated by a linear conjunctive grammar, or, equivalently, accepted by a triangular trellis automaton. This result has several implications on the properties of the languages generated by conjunctive grammars of the general form and on the relationship between the abstract models of parallel computation.

Keywords: Computational complexity; Formal languages; Conjunctive grammars; Trellis automata

1. Introduction

Conjunctive grammars [2] are context-free grammars augmented with an explicit intersection operation, having all rules of the form $A \to \alpha_1 \& \cdots \& \alpha_n$, which basically say that every terminal string that can be generated by every α_i individually can be generated by the nonterminal A as well.

Linear conjunctive grammars [2,4] are a subclass of conjunctive grammars, in which the rules are restricted to be of the form $A \to u_1 B_1 v_1 \& \cdots \& u_n B_n v_n$ or $A \to w$, where A, B_i are nonterminal symbols and u_i, v_i, w are terminal strings. It has recently been proved [5] that linear conjunctive grammars are computationally equivalent to *triangular trellis automata* (TTA) [1], also known as *one-way real-time cellular automata*, which we will define as a quintuple $M = (\Sigma, Q, I, \delta, F)$, where Σ is the input alphabet, Q is the set of states, $I : \Sigma \to Q$ converts each symbol of the input string to a state, thus obtaining the initial ID of the automaton, $\delta : Q \times Q \to Q$ converts a pair of states into a single state, and is used to transform an ID $(q_1, q_2, q_3, \ldots, q_{n-1}, q_n)$ to the ID $(\delta(q_1, q_2), \delta(q_2, q_3), \ldots, \delta(q_{n-1}, q_n))$; this is being done until the string shrinks to one state, when the membership of this single state in F determines whether the input string is accepted.

In this paper we show that a certain P-complete language can in fact be *generated* by a linear conjunctive grammar. This is being done by presenting an encoding of Circuit Value Problem (CVP), the basic P-complete problem, under which it can be solved by the combined force of linear conjunctive grammars, TTA and the known results on these generative devices [4, 5]. Since the general membership problem for linear conjunctive grammars (as well as that for conjunctive grammars of the general form) is also known to be P-complete [3], this language can be said to be compu-

E-mail address: okhotin@cs.queensu.ca (A. Okhotin).

0020-0190/02/$ – see front matter
doi:10.1016/S0020-0190(02)00511-2

tationally as hard as the whole family, which justifies the title of *the hardest linear conjunctive language*. This result also has some immediate theoretical implications, which are discussed in the last section of this paper.

2. The problem and its encoding

Let $(C_1, \ldots, C_n)$ $(n \geqslant 1)$ be a circuit with input variables $x_1, \ldots, x_m$ $(m \geqslant 1)$, where each gate C_k is one of the following: (i) an input x_i $(1 \leqslant i \leqslant m)$; (ii) negation of a preceding gate: $\neg C_i$ $(i < k)$; (iii), (iv) conjunction or disjunction of two preceding gates: $C_i \wedge C_j$ or $C_i \vee C_j$ $(i < j < k)$. The gate C_n is called the output of the circuit. The Circuit Value Problem (CVP) is stated as follows: given a circuit and a Boolean vector of input values $(\sigma_1, \ldots, \sigma_m)$ assigned to the variables, determine, whether the circuit evaluates to true on this vector. The pair (*circuit, vector of input values*) is called an instance of CVP.

Let us develop one particular encoding for instances of CVP as strings over a finite alphabet. Define the alphabet $\Sigma = \{a, b, c_r, c_\wedge, c_\vee, c_w, c_{w\neg}, c_{w0}, c_{w1}\}$, in which a is used to specify the total number of gates in the circuit in unary notation, b is used to refer to the gates of the circuit by writing their numbers in unary notation, c_r is a read command, $c_\wedge$ and $c_\vee$ are commands for computing conjunction and disjunction, respectively, and c_w, $c_{w\neg}$, c_{w0} and c_{w1} (collectively referred to as *write commands*) mean "write", "write negation", "write zero" and "write one", respectively.

Let $((C_1, \ldots C_n), (\sigma_1, \ldots, \sigma_m))$ be an instance of Circuit Value Problem. For each gate C_k, define its literal representation $\omega(C_k)$: $\omega(C_k = x_i)$ equals $b^{k-1}c_{w0}$ if $\sigma_i = 0$ and $b^{k-1}c_{w1}$ if $\sigma_i = 1$; $\omega(C_k = \neg C_i)$ is defined as $b^{i-1}c_r b^{k-i-1}c_{w\neg}$; $\omega(C_k = C_i \wedge C_j)$ is $b^{i-1}c_r b^{j-i-1}c_\wedge b^{k-j-1}c_w$; similarly, $\omega(C_k = C_i \vee C_j) = b^{i-1}c_r b^{j-i-1}c_\vee b^{k-j-1}c_w$ (note that $|\omega(C_k)| = k$ for every kth gate regardless of its type). While a gate can be viewed as an *assignment statement*, its literal representation is a *microprogram* in a very low level language, which implements the assignment. Finally, define the encoding of the whole instance of the problem as

$$a^n \cdot \omega(C_1) \cdot \cdots \cdot \omega(C_n) \in \Sigma^*. \tag{1}$$

Example 1. Consider the circuit $\mathcal{C} = (C_1 = x_1, C_2 = x_2, C_3 = C_1 \wedge C_2, C_4 = \neg C_2, C_5 = C_3 \vee C_4)$ depending on the variables x_1, x_2. It can easily be checked that the circuit implements implication $f(x_1, x_2) = x_2 \to x_1$. Let $x_1 = 1$ and $x_2 = 0$, and consider the instance of CVP $(\mathcal{C}, (1, 0))$.

The literal representations of the circuit's gates are $\omega(C_1 = x_1) = c_{w1}$, $\omega(C_2 = x_2) = bc_{w0}$, $\omega(C_3 = C_1 \wedge C_2) = c_r c_\wedge c_w$, $\omega(C_4 = \neg C_2) = bc_r bc_{w\neg}$ and $\omega(C_5 = C_3 \vee C_4) = bbc_r c_\vee c_w$; consequently, the whole instance of CVP is encoded as

$$aaaaa\, c_{w1}\, bc_{w0}\, c_r c_\wedge c_w\, bc_r bc_{w\neg}\, bbc_r c_\vee c_w. \tag{2}$$

The encoding of CVP presented in this section is different from the traditional encoding in its use of unary notation for numbers. However, the numbers thus notated refer to gates, and thus are bounded by the size of the input; consequently, the descriptions of circuits using binary notation can be converted to unary notation with no more than quadratic blowup, and this conversion can be easily done in logarithmic space, which proves that the language of yes-instances of CVP under the given encoding remains a P-complete language.

3. Solution for the problem

The first thing to do is to check whether the given string is syntactically correct—i.e., whether it represents an instance of CVP encoded using the method of Section 2. This task is already less trivial than could be expected, because even this language is non-context-free (indeed, if we suppose that it is context-free, then its homomorphic image under $h(\text{every symbol}) = a$, which equals $\{a^{n(n+3)/2} \mid n \geqslant 1\}$, is also context-free, which is known to be untrue).

We have to check that the string is of the form (1), or, equivalently, of the form $a^n u_1 u_2 \ldots u_n$, where every u_k is a string of exactly k symbols of the form

$$b^* c_{w0}, \quad b^* c_{w1}, \quad b^* c_r b^* c_{w\neg}, \quad b^* c_r b^* c_\wedge b^* c_w,$$
$$\text{or} \quad b^* c_r b^* c_\vee b^* c_w. \tag{3}$$

The last (and only the last) symbol of each u_k is one of $\{c_w, c_{w\neg}, c_{w0}, c_{w1}\}$ and is called *the write command of the gate*.

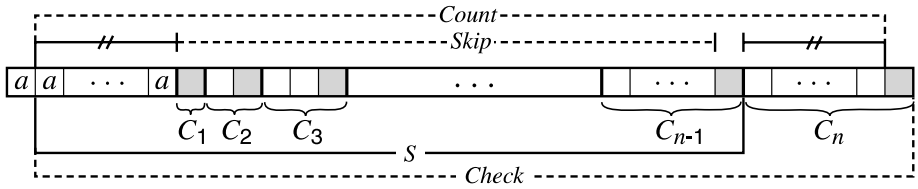

Fig. 1. Checking the syntax by a linear conjunctive grammar.

The set of valid literal representations of circuits can be defined inductively: a string $w \in \Sigma^*$ is a syntactically correct description of a circuit if and only if (*Basis*) $w \in \{ac_{w0}, ac_{w1}\}$, or (*Induction step*) the string w can be factorized as $w = a^n uv$, where $n \geqslant 2$, $u, v \in (\Sigma \setminus \{a\})^+$, $|v| = n$, u ends with a write command, v is of the form (3) and $a^{n-1}u \in \Sigma^*$ is a syntactically correct description of a circuit. It is not hard to construct a linear conjunctive grammar that will generate the set of strings conforming to this definition:

$$\mathtt{S} \to a\ \mathtt{Count}\ s_{\text{write}}\ \&\ a\ \mathtt{Check}$$
$$(s_{\text{write}} \in \{c_{w0}, c_{w1}, c_{w\neg}, c_w\}) \tag{4a}$$
$$\mathtt{S} \to a\ s_{w\,\text{digit}}$$
$$(s_{w\,\text{digit}} \in \{c_{w0}, c_{w1}\}) \tag{4b}$$
$$\mathtt{Count} \to a\ \mathtt{Count}\ s_{\text{inside}}$$
$$(s_{\text{inside}} \in \{b, c_\vee, c_{w\wedge}, c_r\}) \tag{4c}$$
$$\mathtt{Count} \to \mathtt{Skip}\ s_{\text{write}}$$
$$(s_{\text{write}} \in \{c_{w0}, c_{w1}, c_{w\neg}, c_w\}) \tag{4d}$$
$$\mathtt{Skip} \to \mathtt{Skip}\ s_{\text{not-}a}$$
$$(s_{\text{not-}a} \in \{b, c_\vee, c_{w\wedge}, c_r, c_{w0}, c_{w1}, c_{w\neg}, c_w\}) \tag{4e}$$
$$\mathtt{Skip} \to \varepsilon \tag{4f}$$
$$\mathtt{Check} \to \mathtt{Check}_1\ c_{w0} \mid \mathtt{Check}_1\ c_{w1} \mid \mathtt{Check}_2\ c_{w\neg} \mid \mathtt{Check}_3\ c_w \tag{4g}$$
$$\mathtt{Check}_1 \to \mathtt{Check}_1\ b \mid \mathtt{S} \tag{4h}$$
$$\mathtt{Check}_2 \to \mathtt{Check}_2\ b \mid \mathtt{Check}_1\ cr \tag{4i}$$
$$\mathtt{Check}_3 \to \mathtt{Check}_3\ b \mid \mathtt{Check}_2\ c_\vee \mid \mathtt{Check}_2\ c_\wedge \tag{4j}$$

The rules (4b) define one-gate circuits ac_{w0} and ac_{w1}, while the rules (4a) implement the induction step as stated above, with the nonterminal Count comparing the number of symbols in the last gate to the number of prefix a's, and with the nonterminal Check simulating a finite automaton to determine whether the last gate is of the form (3), and then using the nonterminal S to ensure that the last $n-1$ a's and the first $n-1$ gates form a correct description of a circuit. This is illustrated in Fig. 1, where the grey cells denote the write commands from the set $\{c_w, c_{w\neg}, c_{w0}, c_{w1}\}$.

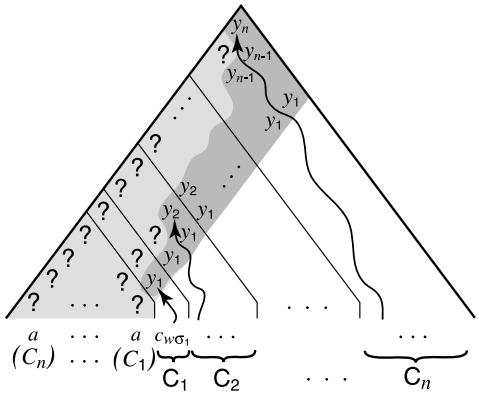

Fig. 2. Overview of the computation on a circuit description $a^n\omega(C_1)\ldots\omega(C_n)$.

Now it suffices to solve the following problem: given a description of a circuit and assuming that it is syntactically valid, determine whether it evaluates to 1. It turns out that this task can be solved by a TTA. Let us explain the main idea of the construction of such an automaton; the interested reader can find the technical details in [6]. The information flow in the automaton's computation is outlined in Fig. 2: the grey band formed by n diagonals starting from the prefix a^n of the whole input string of length $O(n^2)$ is used to store the computed values of the gates, and the description of every gate is used as a microprogram implementing a single assignment statement. The value of ith gate of a circuit $w = a^n \cdot \omega(C_1) \cdot \cdots \cdot \omega(C_n)$ is stored in the diagonal corresponding to the $(n-i+1)$th symbol a; initially it is set to "?", and once the value of the gate is computed, the diagonal keeps reproducing this computed value (0 or 1) till the end of the computation; the subsequent gates have read-only access to this value.

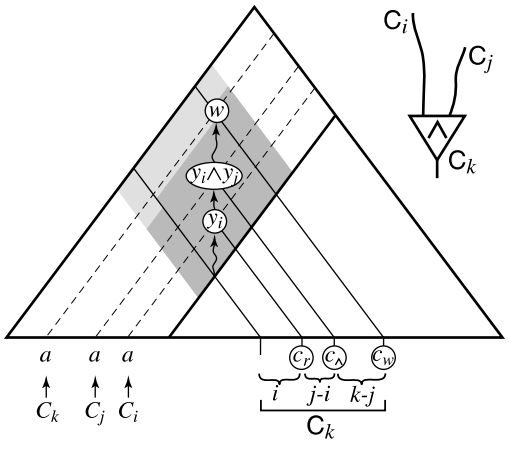

Fig. 3. Operation of an individual conjunction gate $C_k = C_i \wedge C_j$.

Let us consider how the individual gates work, using a conjunction gate $C_k = C_i \wedge C_j$ $(i < j < k)$ as an example. Fig. 3 shows some middle point of the grey band of circuit values, which corresponds to the gate C_k. At this point the values of the gates $C_1, \ldots, C_{k-1}$ have already been computed, and the area shown in the figure is expected to use these values to compute the value of C_k.

The literal representation of the gate C_k, $b^{i-1}c_r b^{j-i-1}c_\wedge b^{k-j-1}c_w$, could be, as mentioned above, viewed as a microprogram implementing the assignment statement $C_k = C_i \wedge C_j$. The substring $b^{i-1}c_r$, i symbols long, is used to find the value of the gate C_i in the grey band and to remember its value. The substring $b^{j-i-1}c_\wedge$ seeks the diagonal corresponding to the gate C_j, at the same time carrying the value of the gate C_i; its last symbol $c_\wedge$ instructs the automaton to read the value of the gate C_j and to compute its conjunction with the remembered value of the gate C_i. This value is carried on by the substring $b^{k-j-1}c_w$, which finds the gate C_k, and replaces the question mark there with the computed value $C_i \wedge C_j$.

Let us now construct a TTA to perform this kind of computation. Define $M = (\Sigma, Q, I, \delta, F)$, where the set of states Q is

$$\underbrace{\{?, 0, 1\}}_{\text{Digit}} \times \Big(\underbrace{\{-, r, \vee, \wedge, w0, w1, w\neg, w\}}_{\text{Commands}} \cup \underbrace{\{\nwarrow, \nwarrow 0, \nwarrow 1, \nearrow, \nearrow 0, \nearrow 1\}}_{\text{Arrows}} \cup \{\sqcup\}\Big). \tag{5}$$

The first component of every pair, *the digit*, can be 0 or 1 only in the diagonals forming the grey band in Figs. 2 and 3, once the value of the corresponding gates is computed (shown in dark grey); all other cells have ? here.

The second component can be a command, an arrow or a space. The eight possible *commands* originate from the input symbols b, c_r, $c_\vee$, $c_\wedge$, c_{w0}, c_{w1}, $c_{w\neg}$ and c_w, respectively; they are propagated in the left direction until executed, where execution results in conversion to an arrow or in alteration of the digit in a diagonal. The *arrows* organize the data flow necessary to execute commands: their intended direction is upward, but since TTA do not support direct upward communication, it is being implemented by a pair of a left-up and a right-up arrow. An arrow can optionally carry some value, obtained by an earlier read command, a conjunction command or a disjunction command, and intended to be used by some later conjunction, disjunction, write or write negation command. The states with a *space* as a second component are called the *unmarked states* and do not have any task to perform besides holding the digit and propagating it to the right; these states can only appear in the diagonals corresponding to the values of the gates.

We shall use the following compact notation for the states from Q: the states of the form $(?, \text{command})$ will be denoted by the name of the command alone—i.e., $w\neg$ means $(?, w\neg)$. The states that have space as the second component will be denoted by the first component alone: ?, 0 and 1 mean $(?, \sqcup)$, $(0, \sqcup)$ and $(1, \sqcup)$, respectively. The states of the form $(0, \text{command})$ and $(1, \text{command})$ will be denoted as the number with the name of the command as a subscript: $0_\wedge$ means $(0, \wedge)$ and 1_{w0} means $(1, w0)$. The states that have an arrow as the second component will be denoted with their second component as a superscript: $0^\nearrow$ and $1^{\nwarrow 0}$ mean $(0, \nearrow)$ and $(1, \nwarrow 0)$, respectively.

A state is called *an activator* if one of the following holds:

(i) it is an unmarked state,
(ii) it is marked with a right arrow, or
(iii) it holds the digit 0 or 1, and it is marked with a write command ($w0$, $w1$, w or $w\neg$).

The full list of activators is $\{?, 0, 1, 0_{w0}, 1_{w0}, 0_{w1}, 1_{w1}, 0_w, 1_w, 0_{w\neg}, 1_{w\neg}, ?^\nearrow, 0^\nearrow, 1^\nearrow, ?^{\nearrow 0}, 0^{\nearrow 0}, 1^{\nearrow 0}, ?^{\nearrow 1},$

$0^{\nearrow 1}$, $1^{\nearrow 1}\}$. Whenever an activator q' and a command q'' meet—i.e., whenever $\delta(q', q'')$ is computed for an activator q' and a state q'' marked with a command—the command is *executed*:

- If q'' is marked with a write command ($w0$, $w1$, w or $w\neg$), then it can be proved that the digit in q' must be ?. Additionally, it can be proved that if q'' is marked with w or $w\neg$, then q' is marked with a right arrow carrying a digit. The state $\delta(q', q'')$ is then defined as (0, ␣), if q'' is marked with $w0$; as (1, ␣), if q'' is marked with $w1$; as (the digit on the arrow in q', ␣), if q'' is marked with w; and as (negation of the digit on the arrow in q', ␣), if q'' is marked with $w\neg$. Note that this is the only case when the digit in $\delta(q', q'')$ differs from the digit in q'.
- If q'' is marked with a read command r, then $\delta(q', q'')$ retains the digit from q' and is marked with a left arrow carrying the same digit from q'—i.e., $\delta(q', q'')$ can be $0^{\nwarrow 0}$ or $1^{\nwarrow 1}$ depending on q'.
- If q'' is marked with a dash command (meaning "proceed to the next row"), then $\delta(q', q'')$ retains the digit from q' and, if q' is marked with a right arrow carrying some digit, then $\delta(q', q'')$ is marked with a left arrow carrying the same digit; otherwise, if q' is marked with something else, then $\delta(q', q'')$ is marked with a left arrow carrying nothing.
- If q'' is marked with a $\wedge$ or $\vee$ command, then q' can be proved to be marked with a right arrow carrying a digit. $\delta(q', q'')$ retains the digit from q' and is marked with a left arrow carrying the digit computed by the command: if q'' is marked with $\wedge$ command, then the arrow in $\delta(q', q'')$ is marked with the conjunction of the first component of q' and the digit carried by the arrow in q'; if q'' is marked with $\vee$ command, then the arrow in $\delta(q', q'')$ is marked with the disjunction of the first component of q' and the digit carried by the arrow in q'.

Another case is the so-called *turning of the arrow*: if q'' is marked with a left arrow (optionally carrying some digit), then the second component of $\delta(q', q'')$ is a right arrow loaded with the same cargo as the left arrow of q''; the digit in $\delta(q', q'')$ is inherited from q'. In all other cases the transition $\delta(q', q'')$ is made by reduplicating the digit from q' and, if q'' is marked with a command, by reduplicating this command, otherwise by producing an unmarked state.

The initial function $I : \Sigma \to Q$ maps a to "?", b to "$-$" and every command $c_{\ldots}$ to the state with the same name as the subscript of c (e.g., $I(c_{w0}) = w0$). The sole accepting state is (1, ␣), or 1 in the short notation.

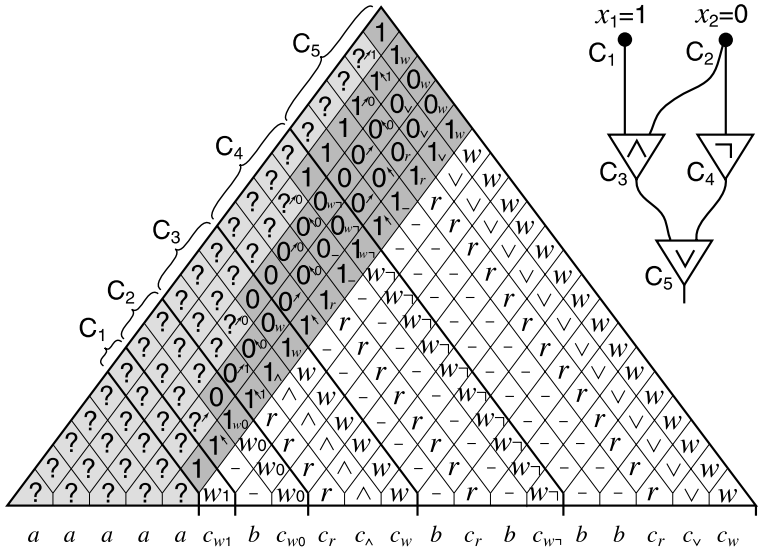

Fig. 4. Sample computation of the automaton for CVP.

Consider the circuit from Example 1 in Section 2, shown in the top right corner of Fig. 4, and the input values $(\sigma_1, \sigma_2) = (1, 0)$. This instance of CVP is encoded as (2), and the computation of the automaton on this string is given in Fig. 4; it is an accepting computation, because 1 is an accepting state.

Let us now put together the results obtained so far using the following known properties of linear conjunctive languages:

Proposition 1 [5]. *A language $L \subseteq \Sigma^+$ is generated by a linear conjunctive grammar if and only if it is accepted by a TTA.*

Proposition 2 [4]. *The class of linear conjunctive languages is closed under all set-theoretic operations.*

Having constructed a linear conjunctive grammar for checking syntax and a TTA for computing the value of a properly described circuit, we can use Proposition 1 to find out that both languages belong to the class of linear conjunctive languages. It can be observed that the language of CVP is the intersection of these two languages, which is linear conjunctive by Proposition 2. Therefore, there exist both a linear conjunctive grammar and a TTA solving the entire circuit value problem. Full technical details of the construction are provided in [6].

4. Theoretical implications

We have demonstrated that linear conjunctive grammars can generate a language, which is, as argued in the Introduction, computationally as hard as the whole family; although the same holds for context-sensitive grammars, this seems to be the first result of this kind for practically useful families of grammars.

Besides partially satisfying one's curiosity on the generative power of linear conjunctive grammars and TTA, the result of this paper has several theoretical implications. One concerns the complexity of parsing for conjunctive grammars. Although most of the context-free parsing algorithms (LL, generalized LR and several tabular algorithms) have already been extended for the case of conjunctive grammars without increasing their complexity, logarithmic-time parallel parsing algorithms [8] defied generalization. Now we see that the existence of such a generalization would imply that some P-complete problem can be solved in NC [7,9] (NLOGSPACE $\subseteq$ NC $\subseteq$ P), and conjecture that no such parsing algorithm exists:

Proposition 3. *If* NC $\neq$ P, *then logarithmic-time parallel parsing algorithms for context-free grammars cannot be generalized for conjunctive grammars.*

Second, we can now return to the question (raised in [2]) on whether conjunctive languages are closed under ε-free homomorphism. Define

$$\Gamma = \big(\Sigma \setminus \{c_{w0}, c_{w1}\}\big) \cup \{c_{\text{in}}\}$$

and consider the homomorphism $h : \Sigma^+ \to \Gamma^+$ that maps both c_{w0} and c_{w1} to c_{in}, and maps other symbols from Σ to themselves. Clearly, h turns the circuit value problem into the circuit satisfiability problem, which implies the following:

Theorem 1. *The language $h(L_{\text{CVP}})$ is an* NP*-complete language.*

Corollary 1. *If* P $\neq$ NP, *then the family of conjunctive languages is not closed under letter-to-letter homomorphism and under ε-free homomorphism.*

One more theoretical implication of the result of this paper concerns the abstract models of parallel computation. Both TTA (also under the name of one-way real-time cellular automata) and circuits of polylogarithmic depth are sometimes used as models of parallel computation, and the latter defines the complexity class NC. As we have just proved, TTA are powerful enough to solve complete problems in the class P, which are, on the other hand, believed not to be solvable by polylogarithmic-depth circuits (the NC $\neq$ P assumption). This suggests that these two models of parallel computation are very different in power, and triangular trellis automata can solve much harder problems.

References

[1] C. Choffrut, K. Culik II, On real-time cellular automata and trellis automata, Acta Inform. 21 (1984) 393–407.

[2] A. Okhotin, Conjunctive grammars, J. Automat. Languages Combin. 6 (4) (2001) 519–535.

[3] A. Okhotin, A recognition and parsing algorithm for arbitrary conjunctive grammars, Theoret. Comput. Sci., to appear. doi: 10.1016/S0304-3975(02)00853-8.
[4] A. Okhotin, On the closure properties of linear conjunctive languages, Theoret. Comput. Sci. 299 (2003) 663–685.
[5] A. Okhotin, Automaton representation of linear conjunctive languages, in: Developments in Language Theory, Proc. DLT 2002, Kyoto, Japan, Lecture Notes in Comput. Sci., Springer, Berlin, to appear.
[6] A. Okhotin, A linear conjunctive grammar for the circuit value problem, Tech. Rept. 2002-460, School of Computing, Queen's University, Kingston, ON, 2002.
[7] Ch.H. Papadimitriou, Computational Complexity, Addison-Wesley, Reading, MA, 1994.
[8] W. Rytter, On the recognition of context-free languages, in: 5th Symposium on Fundamentals of Computation Theory, in: Lecture Notes in Comput. Sci., Vol. 208, Springer, Berlin, 1985, pp. 315–322.
[9] H. Vollmer, Introduction to Circuit Complexity: A Uniform Approach, Springer, Berlin, 1999.