Simulating Boolean circuits by finite splicing

Katrin Erk
Programming Systems Lab
Universität des Saarlandes, Saarbrücken, Germany
erk@ps.uni-sb.de

Abstract- As a computational model to be simulated in a DNA computing context, Boolean circuits are especially interesting because of their parallelism. Simulations in concrete biochemical computing settings have been given by [OR96] and [AD97]. In this paper, we show how to simulate Boolean circuits by finite splicing systems, an abstract model of enzymatic recombination ([Hea87], [Gat94], [Pău96a]). We argue that using an abstract model of DNA computation as a basis leads to simulations of greater clarity and generality. In our construction, the running time of the simulating system is proportional to the depth, and the use of material is proportional to the size of the Boolean circuit simulated. However, the rules of the simulating splicing system depend on the size of the Boolean circuit, but not on the connectives used.

Keywords: DNA computing, splicing, Boolean circuits, parallelism

1 Introduction

Boolean circuits are graphs that can represent any Boolean function. The nodes of a Boolean circuit are labeled with input variables and Boolean connectives. Its directed edges describe the information flow.

In the context of DNA computing, Boolean circuits are interesting for several reasons. One is that they are a model of parallel computation: The value of each node can be computed as soon as the values of its input nodes are established (and sometimes sooner), so a simulation of Boolean circuits can exploit the inherent parallelism of DNA computations. Another point is their importance and simplicity — relatively simple DNA computing systems suffice for simulating them.

Boolean circuits have been simulated in a DNA environment by [OR96] and [AD97]. [OR96] use annealing, ligation, separation by size (gel electrophoresis), selective amplification (PCR), and cleaving by restriction enzymes to simulate AND and OR gates. They get negation by having, for n input variables $x_1, \dots, x_n$, circuits with input gates labeled $x_1, \dots, x_n, \overline{x}_1, \dots, \overline{x}_n$. [AD97] simulate NAND gates in a less expensive translation which also avoids the error–prone PCR operation. In this paper, we describe another simulation of Boolean circuits, but whereas [OR96] and [AD97] both use concrete biochemical settings, this paper takes an abstract model of DNA computation, splicing systems, as a basis.

The field of DNA computing has both practical and theoretical aspects. On the ”practical side”, algorithms are designed for DNA systems capable of executing some concrete set of biochemical operations, and laboratory experiments are conducted to test the feasibility of these algorithms. Theoretical studies of DNA computing formalize biochemical operations in abstract models, both as a mathematical foundation of DNA computation and as novel models of computation that are interesting in their own right. If a system of this latter kind is applied, it is mostly to the simulation of a standard model of computation in order to establish the computational power of the theoretical DNA computing system. There is little exchange between the two areas.

We propose the use of abstract models of DNA computation as a basis for the formulation of algorithms, for the sake of greater clarity and generality. Concrete ”biochemical algorithms” are specific to the set of bio–operations used: If an operation used in an algorithm turns out to be particularly error–prone and is to be avoided, the algorithm has to be redesigned from scratch. But of course, if we use an abstract model as a basis for an algorithm, it has to be sufficiently close to biochemical reality for the algorithm to be of practical interest. We think that finite splicing systems meet this requirement.

Splicing systems model the sequence–specific cleaving of DNA molecules by certain enzymes and the re–connecting of cleft parts: A splicing system initially contains a language over some finite alphabet V, modelling the DNA molecules initially in the test tube. If two words v, w present in the ”test tube”, i.e. the language, contain subsequences specified in a splicing rule, they are both cut, and the prefix of v is concatenated to the suffix of w and vice versa. After the splicing operation, the language contains v, w as well as the two new words resulting from the splicing. Splicing systems are parallel in the sense that the biochemical operation they model is parallel: All the molecules that can be cut by the enzymes present in the test tube are cut at the same time. Up to now, splicing systems have been investigated mainly with respect to the formal languages they are capable of generating. Our focus is different: We require that our construction exploit the parallelism of splicing systems and always compute the value of the simulated Boolean circuit as a single splicing word, in finite time.

In our construction, we use finite splicing systems to simulate Boolean circuits. The number of parallel computation steps the splicing systems needs is proportional to the depth of the Boolean circuit, while the amount of material (the size of the initial language and the number of splicing rules) is proportional to the circuit size. But while the number of splicing rules depends on the size of the Boolean circuit, it is independent of the Boolean connectives used, which take effect solely in the initial language.

This paper is organized as follows: Section 2 introduces splicing systems. In section 3, Boolean circuits are defined. In section 4 we describe our splicing simulation of Boolean circuits. Section 5 concludes.

2 Splicing Systems

A single–stranded DNA molecule consists of a sugar–phosphate backbone on which four different bases, adenine, cytosine, guanine and thymine, abbreviated A, C, G, T, are arranged in arbitrary order. Two single strands can combine to form a double–stranded DNA molecule if their base sequences are complementary: A and T are complementaries, as well as C and G. *Restriction enzymes* cut double-stranded DNA into pieces whenever they encounter certain triggering subsequences of DNA, their *sites*. Many of these enzymes leave single-stranded overhangs. Figure 1 shows a schematic depiction of two restriction enzymes cutting DNA molecules.

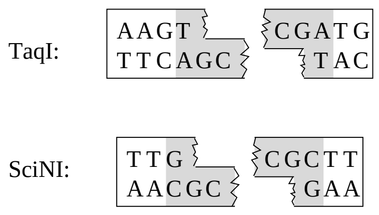

Figure 1: Two restriction enzymes cutting two DNA molecules. The shaded areas are the enzymes' sites.

DNA molecules with single–stranded overhangs can be connected if the overhangs are complementary and of the same orientation (which depends on the molecules that the sugar–phosphate backbones of the overhangs end in), if a *ligase* is present. For the molecules cleft in figure 1, either the two pieces of the same molecule can reconnect, or the pieces can be combined crosswise, as the overhangs match. Figure 2 shows this second case. (See e.g. [Str91] for a detailed description.)

These two operations together, the sequence–specific cleaving of molecules and the ligating of matching pieces, are called *recombination*. This biochemical operation was formalized in splicing systems by [Hea87]. In splicing systems, DNA molecules are modelled by words, and recombination is modelled by splicing rules that allow for suffix exchange

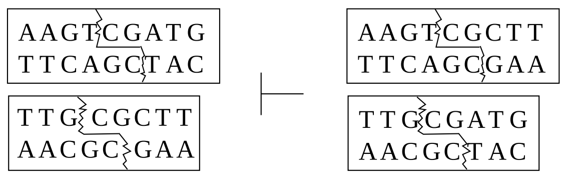

Figure 2: The molecules cut in figure 1 combined crosswise

between words. Head's definition was simplified by Gatterdam in [Gat94] and further generalized to encompass infinite rule sets by Păun in [Pău96a]. In this paper, we use this latter definition:

Definition 2.1 Let V be a finite alphabet not containing the symbols $\#, \$$.

A **splicing system** over V is a construct $S = (V, I, R)$, where

- $I \subseteq V^*$ is the initial language and
- $R \subseteq V^*\#V^*\$V^*\#V^*$ is the set of splicing rules.

A rule $u_1\#u_2\$u_3\#u_4$ is often written as $\begin{array}{c|c} u_1 & u_2 \\ \hline u_3 & u_4 \end{array}$.

A splicing rule $r = u_1\#u_2\$u_3\#u_4$ is applicable to two words $v, w \in V^*$ if there are words $v', v'', w', w'' \in V^*$ such that $v = v'u_1u_2v''$ and $w = w'u_3u_4w''$. Applying the rule produces the two new words $\overline{v} = v'u_1u_4w''$ and $\overline{w} = w'u_3u_2v''$. We also write

$$(v, w) \vdash (\overline{v}, \overline{w}).$$

The **language of a splicing system S, $L(S)$**, is the smallest set L such that $I \subseteq L$, and if $v, w \in L$ and there is a splicing rule r in S such that $(v, w) \vdash_r (\overline{v}, \overline{w})$, then $\overline{v}, \overline{w} \in L$.

We denote the empty word by ε.

A splicing sytem assumes an unlimited supply of each word and cumulatively adds splicing results until it is saturated.

To characterize the computational power of splicing systems, two points are especially interesting: Splicing systems with a finite initial language and a regular rule set already generate all recursively enumerable languages ([Pău96b]), whereas splicing systems of finite components can only generate regular languages ([CH91], [Pix95], [HPP96]).

Example 2.2 A biochemical recombination system containing the enzymes TaqI and SciNI described above could be modelled using the alphabet $\{{A \atop T}, {C \atop G}, {G \atop C}, {T \atop A}\}$ and the splicing rules

$$\begin{array}{c|c} {T \atop A}{C \atop G}{G \atop C} & {A \atop T} \\ \hline {T \atop A}{C \atop G}{G \atop C} & {A \atop T} \end{array}, \begin{array}{c|c} {G \atop C}{C \atop G}{G \atop C} & {C \atop G} \\ \hline {G \atop C}{C \atop G}{G \atop C} & {C \atop G} \end{array}, \begin{array}{c|c} {T \atop A}{C \atop G}{G \atop C} & {A \atop T} \\ \hline {G \atop C}{C \atop G}{G \atop C} & {C \atop G} \end{array}$$

Two molecule pieces can be ligated if they have both been cut by TaqI, if they have both been cut by SciNI or if one has been cut by TaqI and the other by SciNI.

Splicing systems as defined in [Pău96a] abstract from many facts of recombination. For example, a splicing rule specifies both words involved in the suffix exchange, while an enzyme only describes one of the two molecules concerned. Recombination systems are reflexive: Two molecules containing the site of TaqI can always recombine, whereas in splicing systems, it does not follow that the rule $u_1\#u_2\$u_1\#u_2$ is present if $u_1\#u_2\$u_3\#u_4$ is. The original definition of splicing systems in [Hea87] is very close to biochemical recombination. There, splicing rules describe just one enzyme each, and for a splicing reaction two splicing rules with matching sites are required; furthermore, these systems are always reflexive. But as shown in [Erk98], splicing systems as defined in [Pău96a] do not possess a greater generative power than the original definition from [Hea87]. So we can choose the more abstract and thus easier to use definition from [Pău96a] without losing reference to biochemical DNA computing systems.

Splicing systems are abstract models of recombination; but that does not mean that a biochemical implementation of a splicing algorithm must use solely recombination operations. Examples of other biochemical operations suitable for implementing splicing steps are annealing or ligating without prior cleaving.

3 Boolean circuits

A Boolean circuit is a directed graph that represents a Boolean function. Its nodes stand for input and output values and for Boolean connectives, and its directed edges show the flow of information. Figure 3 shows an example of a Boolean circuit; nodes with no incoming edges are input nodes labeled with either variables or a constant value 1 or 0, and the node with no outgoing edges is the output node of the circuit.

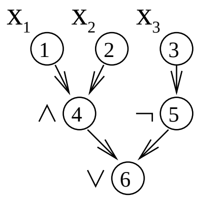

Figure 3: A Boolean circuit representing the Boolean function $f(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee \neg x_3$. Nodes contain their numbers and are annotated with their sorts.

Definition 3.1 A **basis** is a set Ω of Boolean functions $f : \{0,1\}^i \to \{0,1\}$, $i \in \{0,1,2\}$.[1] Ω is called **complete** if all Boolean functions can be expressed by composition of functions from Ω.

A **Boolean circuit with M inputs** $x_1, \ldots, x_M$ **of basis Ω** is a finite directed acyclic graph $G = (V_G, E_G)$, where each node of V_G has an indegree of 0, 1 or 2. The nodes of V_G are also called **gates**. Each gate $v \in V_G$ is assigned a **sort** $s(v)$, where $s(v) \in \Omega \cup \{x_1, \ldots, x_M\}$. If $s(v) \in \{x_1, \ldots, x_M\}$ or $s(v) \in \{0,1\}$, v is called **input** of G. In this case v must have indegree 0. If the sort of v is a Boolean function of arity 1 (2), v must have 1 (2) incoming edges.

Let a topological sorting of the nodes in V_G be given as $v_1, \ldots, v_N$, where $N = |V_G|$. [2] The node v_N is called the **output** of G. (If the circuit computes several Boolean functions at once, it may possess several output nodes. In this case, every node of outdegree 0 is called output.)

The **depth of a node** $v \in V_G$ is the number of nodes in the longest directed path connecting an input of G to v. The **depth of the circuit** G is the depth of its output (if G possesses more than one output node, its depth is the maximum of its output node depths).

An assignment $A : \mathcal{V} \to \{0,1\}$ for some set $\mathcal{V}$ of variables is called *appropriate* for G if it is defined for all variables from $\{x_1, \ldots, x_M\}$. Given an appropriate assignment A, the **truth value of a gate $\mathbf{v_i} \in \mathbf{V_G}, \mathbf{A(v_i)}$**, is defined by induction on its number in the topological sorting:

- If $s(v_i) \in \{x_1, \ldots, x_M\}$, then $A(v_i) = A(s(v_i))$.
- If $s(v_i) \in \{0,1\}$, then $A(v_i) = s(v_i)$.
- If $s(v_i) = f : \{0,1\} \to \{0,1\}$, there is a unique gate v_j with $(v_j, v_i) \in E_G$, thus $j < i$. By induction, the truth value of v_j has already been established, and we set $A(v_i) = f(A(v_j))$.
- If $s(v_j) = f : \{0,1\}^2 \to \{0,1\}$, there are exactly two gates v_j, v_k with $(v_j, v_i), (v_k, v_i) \in E_G$ and $j \le k < i$. We set $A(v_i) = f(A(v_j), A(v_k))$.
- The **truth value of the Boolean circuit G, $\mathbf{A(G)}$**, is $A(v_N)$.

Note that this definition imposes an order on the incoming edges of a gate, namely the topological sorting of the gates: If v_o with sort $s(v_o) = f$ has incoming edges from v_n and v_m where $n \le m$, then $A(v_o) = f(A(v_n), A(v_m))$.

4 A splicing simulation of Boolean circuits

In this section, we use circuits with a single output v_N for clarity, although our construction also works for multiple–output circuits. Let $G = (V_G, E_G)$ be a Boolean circuit with N gates topologically sorted as $v_1, \ldots, v_N$. We simulate G by a splicing system $S = (V, I, R)$, where the description of G is given in the initial language I of the splicing system. We use two kinds of words: gate value words and gate words. There are two possible gate value words for gate v_n, carrying a meaning of either

”$A(v_n)$ has been established as 1”

or

”$A(v_n)$ has been established as 0”.

[1] The two constant functions with no inputs are also written as 0 and 1.

[2] If $|V_G| = N$, a topological sorting enumerates the nodes of V_G as $v_1, \ldots v_N$ in such a way that whenever G contains an edge (v_i, v_j) for $i, j \in \{1, \ldots, N\}$ then $i < j$. v_N is the end of the longest directed path in G.

As splicing proceeds, gate value words for an increasing number of gates from V_G will be present in the language of S, proceeding by the depth of gates, until after a number of parallel steps proportional to the depth of G, the value of v_N is established.

For gates with two inputs, it sometimes suffices to know the value of one of its inputs to determine the value of the gate. For example, if in the circuit in figure 3, v_1 has been assigned the value 0, then the value of v_4 is also 0 independent of v_2. But only if $A(v_1) = A(v_2) = 1$ is v_4 assigned the value 1. In the splicing system, we describe the first case by gate words carrying the meaning

"if $A(v_n) = \left\{ {1 \atop 0} \right\}$ then $A(v_o) = \left\{ {1 \atop 0} \right\}$"

for a gate v_o that has v_n as its input or as one of its inputs. For example, when simulating the circuit of figure 3, for v_4 S contains a gate word denoting "if $A(v_1) = 0$ then $A(v_4) = 0$" (but also "if $A(v_2) = 0$ then $A(v_4) = 0$"). If a gate value word meaning "$A(v_1)$ has been established as 0" is also present, then the two words are spliced, resulting in the new word "$A(v_4)$ has been established as 0". A gate value word stating that $A(v_1) = 1$ would not have any effect if combined with this gate word, it only reacts if $A(v_1) = 0$.

The second case, where the values of both input gates have to be considered, is handled by gate words of the intuitive meaning "if $A(v_n) = \left\{ {1 \atop 0} \right\}$ then if $A(v_m) = \left\{ {1 \atop 0} \right\}$ then $A(v_o) = \left\{ {1 \atop 0} \right\}$". For example, the splicing description of gate v_4 in figure 3 also needs to contain a gate word denoting "if $A(v_1) = 1$ then if $A(v_2) = 1$ then $A(v_4) = 1$". When combined with a gate value word meaning "$A(v_1)$ has been established as 1", it is spliced, producing a gate word "if $A(v_2) = 1$ then $A(v_4) = 1$" — a gate word of the first type, which can be further handled as described above.

We now formalize this idea. With the coding we have just sketched, we can simulate any Boolean function $f : \{0,1\}^i \to \{0,1\}, i \in \{0,1,2\}$, but we only present the ones commonly used. Let $G = (V_G, E_G)$ be a Boolean circuit of M inputs and N gates, with $\Omega = \{0, 1, AND, NAND, OR, NOR, NOT, XOR, EQ, \to, \leftarrow\}$ as its basis. [3] Let a topological sorting of the gates be given as $v_1, \ldots, v_N$. For gates $v \in V$ we set $inp(v) = \{n \in \{1, \ldots, N\} \mid (v_n, v) \in E_G\}$, i.e. $inp(v)$ is the set of input gate numbers for v. We construct, for G, a splicing system $S = (V, I, R)$ where

- $V = \{1, \ldots, N\} \cup \{X, Y, Z, D, t, f, T, F\}$,
- $I = \{XY\} \cup I'$, which is described in figure 4.
- R consists of the rules

  1: $\begin{array}{c|c} XnD & Z \\ \hline Xna & X \end{array}$ for $1 \le n \le N, a \in \{t, f\}$,

  2: $\begin{array}{c|c} tZT & \varepsilon \\ \hline X & Y \end{array}$

  3: $\begin{array}{c|c} fZF & \varepsilon \\ \hline X & Y \end{array}$

We use D to denote that the value of a node has not been established yet, t and f as gate values 1 and 0, respectively, and T (F) for gate words that are triggered if the gate's input is 1 (0). X, Y, Z are markers. Note that the ε in the rules above only specifies that there is no restriction on the first spliced word after the cutting point; it does not mean that tZT or fZF is the end of the word — splicing rules cannot express that.

Gate value words have the form

$$\begin{aligned} [\![\, A(v_n) = 1 \,]\!] &:= XntX, \\ [\![\, A(v_n) = 0 \,]\!] &:= XnfX \end{aligned}$$

for $n \in \{1, \ldots, N\}$. (We use the symbolic form $[\![\ldots]\!]$ for greater readability.) Each gate word has a prefix

$$XnDZ$$

for an $n \in \{1, \ldots, N\}$ to denote that v_n is an input of the gate involved, and that v_n's value is as yet unknown. If a gate value word $[\![\, A(v_n) = 1 \,]\!]$ ($[\![\, A(v_n) = 0 \,]\!]$) is present in the system, the D in the gate word can be replaced by the actual value of v_n by the splicing rule

$$\begin{array}{c|c} XnD & Z \\ \hline Xnt & X \end{array} \; \Big(\begin{array}{c|c} XnD & Z \\ \hline Xnf & X \end{array} \Big),$$

leading to a gate word prefix of $XntZ$ ($XnfZ$).

Gate words covering the first case, where only the value of one input gate is needed, have the form

$$\begin{aligned} [\![\, A(v_n) = 1 \to A(v_o) = 1 \,]\!] &:= XnDZTotX, \\ [\![\, A(v_n) = 1 \to A(v_o) = 0 \,]\!] &:= XnDZTofX, \\ [\![\, A(v_n) = 0 \to A(v_o) = 1 \,]\!] &:= XnDZFotX, \\ [\![\, A(v_n) = 0 \to A(v_o) = 0 \,]\!] &:= XnDZFofX \end{aligned}$$

for $n, o \in \{1, \ldots, N\}$. If the prefix XnD has already been replaced by Xnt or Xnf as described above, and if v_n's value corresponds to the marker T or F, then the suffix of the gate word can be spliced off using the catalytic word XY and the rule

$$\begin{array}{c|c} tZT & \varepsilon \\ \hline X & Y \end{array} \; \Big(\begin{array}{c|c} fZF & \varepsilon \\ \hline X & Y \end{array} \Big).$$

This reaction produces the new word $XotX = [\![\, A(v_o) = 1 \,]\!]$ or $XofX = [\![\, A(v_o) = 0 \,]\!]$.

The second case, where the values of two input nodes are considered, is handled by gate words of the form

$$[\![\, A(v_n) = \left\{ {1 \atop 0} \right\} \to A(v_m) = \left\{ {1 \atop 0} \right\} \to A(v_o) = \left\{ {1 \atop 0} \right\} \,]\!] := XnDZ \left\{ {T \atop F} \right\} mDZ \left\{ {T \atop F} \right\} o \left\{ {t \atop f} \right\} X$$

for $m, n, o \in \{1, \ldots, N\}$ for inputs v_n, v_m of gate v_o. We do not need any new splicing rules to bring about the appropriate reactions. Consider the case of a gate word $[\![\, A(v_1) = 1 \to A(v_2) = 1 \to A(v_4) = 1 \,]\!] = X1DZT2DZT4tX$. If $[\![\, A(v_1) = 1 \,]\!] = X1tX$ or $[\![\, A(v_1) = 0 \,]\!] = X1fX$ is present in the language, the first D of the gate word is replaced by v_1's value. A value for v_2 cannot yet be entered as the gate word does not contain the factor $X2DZ$. In the case of $[\![\, A(v_1) = 1 \,]\!]$, we now have $X1tZT2DZT4tX$ (with $[\![\, A(v_1) = 0 \,]\!]$, the resulting word cannot be spliced further), which the rule $\begin{array}{c|c} tZT & \varepsilon \\ \hline X & Y \end{array}$

[3] $\leftarrow (x_1, x_2) := \to (x_2, x_1)$. This function is introduced because of the order our definition imposes on a gate's inputs (see section 3).

transforms to $[\![A(v_2)=1 \to A(v_4)=1]\!] = X2DZT4tX$, a gate word of the first type.

Figure 4 lists the gate words that I' contains for each Boolean connective: Let $v_o \in V_G, 1 \le o \le N$, such that $s(v_o) \notin \{x_1,\ldots,x_M\}$. If v_o has indegree one, let $inp(v_o)=\{n\}$, and if v_o has indegree two, let $inp(v_o)=\{n,m\}$ with $n \le m$, and let $p \in \{n,m\}$. The left column in figure 4 gives the value of $s(v_o)$, while the right column shows the corresponding words of I'.

If the initial language contains a gate word $[\![A(v_n)=\left\{\begin{smallmatrix}1\\0\end{smallmatrix}\right\} \to A(v_m)=\left\{\begin{smallmatrix}1\\0\end{smallmatrix}\right\} \to A(v_o)=\left\{\begin{smallmatrix}1\\0\end{smallmatrix}\right\}]\!]$, the symmetrical word $[\![A(v_m)=\left\{\begin{smallmatrix}1\\0\end{smallmatrix}\right\} \to A(v_n)=\left\{\begin{smallmatrix}1\\0\end{smallmatrix}\right\} \to A(v_o)=\left\{\begin{smallmatrix}1\\0\end{smallmatrix}\right\}]\!]$ is not needed, as this type of gate word establishes the value of v_o only if both input values are present anyway.

The splicing system S describes the circuit G independently of possible assignments. But if an assignment appropriate to G is added to S's initial language I as a set of gate value words, S computes the value of G under this assignment:

Theorem 4.1 *Let G be a Boolean circuit of N gates and M inputs with $\Omega = \{0,1, AND, NAND, OR, NOR, NOT, XOR, EQ, \to, \leftarrow\}$ as its basis. Then there is a finite splicing system $S=(V,I,R)$ such that for each assignment A appropriate to G there exists a finite set $I_A \subseteq V^*$ and $X,t,f \in V$ such that*

$$\begin{array}{lcl} XNtX \in L\big((V, I\cup I_A, R)\big) & \iff & A(G)=1 \text{ and} \\ XNfX \in L\big((V, I\cup I_A, R)\big) & \iff & A(G)=0. \end{array}$$

Proof: Let G and S be defined as above, and let $x_1,\ldots x_M$ be the inputs of G. Let A be an assignment appropriate to G, and let $V_G^I = \{v \in V_G \mid s(v) \in \{x_1,\ldots,x_M\}\}$ be the inputs of G. Then A is translated into the set $I_A = \{XntX \mid v_n \in V_G^I \wedge A(v_n)=1\} \cup \{XnfX \mid v_n \in V_G^I \wedge A(v_n)=0\}$.

"$\Leftarrow$" We prove that for each node $v_o \in V_G, 1 \le o \le N$, $A(v_o)=1$ (0) $\Longrightarrow XotX$ $(XofX) \in L\big((V, I\cup I_A, R)\big)$. We use induction on the depth k of v_o.

$k=0$**:** Let $A(v_o)=1$ (0). Either $s(v_o) \in \{x_1,\ldots,x_M\}$, so $XotX$ $(XofX) \in I_A$, or $s(v_o)=1$ (0), then $XotX$ $(XofX) \in I'$.

$k \Rightarrow k+1$**:** $inp(v_o) \neq \emptyset$. We only consider the case $s(v_o)=AND$, for the other cases splicing proceeds analogously.

If $n \in inp(v_o)$ and $A(v_n)=0$, then $XnfX \in L\big((V, I\cup I_A, R)\big)$ by the inductive hypothesis. For v_o, I' contains the gate word $XnDZFofX$. The following computation is valid in S: [4]

$$\begin{array}{rl} (XnD|ZFofX, Xnf|X) \vdash_1 & (XnfZFofX, \\ & XnDX) \\ (XnfZF|ofX, X|Y) \quad \vdash_3 & (XofX, XnfZFY) \end{array}$$

[4]We annotate splicing operations with the rule type used. We mark the place at which splicing occurs by a $|$.

If $inp(v_o)=\{n,m\}$ and $A(v_n)=A(v_m)=1$, then $XntX, XmtX \in L\big((V, I\cup I_A, R)\big)$ by the inductive hypothesis. Let $n \le m$. Then I' contains the gate word $XnDZTmDZTotX$, and S can compute

$$\begin{array}{lcl} (XnD|ZTmDZTotX, & \vdash_1 & (XntZTmDZTotX, \\ Xnt|X) & & XnDX) \\ (XntZT|mDZTotX, & \vdash_2 & (XmDZTotX, \\ X|Y) & & XntZTY) \\ (XmD|ZTotX, & \vdash_1 & (XmtZTotX, \\ Xmt|X) & & XmDX) \\ (XmtZT|otX, X|Y) & \vdash_2 & (XotX, XmtZTY) \end{array}$$

"$\Rightarrow$" As we have seen in the previous part of the proof, each splicing operation produces, besides the intended result, one "garbage string". We have to show that these side effects do not interfere with the computation.

Application of a rule $\dfrac{XnD \mid Z}{Xna \mid X}$ produces garbage strings $XnDX$, which cannot undergo any further reaction. Rules 2 and 3 leave words $XntZTY$ and $XnfZFY$. They can again be spliced by rules 2 and 3 without generating anything new: $(XntZT|Y, X|Y) \vdash_2 (XntZTY, XY)$ and analogous for rule 3.

If a value $[\![A(v_n)=1]\!]$ ($[\![A(v_n)=0]\!]$) is entered into a gate word starting with $XnDZF$ ($XnDZT$), the result has a prefix $XntZF$ ($XnfZT$). Such a word cannot be spliced any further. ∎

Example 4.2 For the boolean circuit in figure 3, I' contains the following gate words:

for v_4:

$$\begin{array}{lcl} [\![A(v_1)=0 \to A(v_4)=0]\!] & = & X1DZF4fX \\ [\![A(v_2)=0 \to A(v_4)=0]\!] & = & X2DZF4fX \\ [\![A(v_1)=1 \to A(v_2)=1 \to A(v_4)=1]\!] & & \\ & = & X1DZF2DZT4tX \end{array}$$

for v_5:

$$\begin{array}{lcl} [\![A(v_3)=1 \to A(v_5)=0]\!] & = & X3DZT5fX \\ [\![A(v_3)=0 \to A(v_5)=1]\!] & = & X3DZF5tX \end{array}$$

for v_6:

$$\begin{array}{lcl} [\![A(v_4)=1 \to A(v_6)=1]\!] & = & X4DZT6tX \\ [\![A(v_5)=1 \to A(v_6)=1]\!] & = & X5DZT6tX \\ [\![A(v_4)=0 \to A(v_5)=0 \to A(v_6)=0]\!] & & \\ & = & X4DZF5DZF6fX \end{array}$$

Assume an assignment A with $A(x_1)=1, A(x_2)=0$ and $A(x_3)=0$. Then this assignment is expressed by $I_A = \{X1tX, X2fX, X3fX\}$, which is part of the initial splicing language. R comprises 14 rules: 12 to exchange D for t or f, and 2 for transforming enabled gate words.

We only show how the value of v_4 under A is computed; the rest of the splicing proceeds analogously. Each paragraph constitutes one parallel computation step. A gate value word $X4tX$ for v_4 is computed in two parallel steps.

1:	$\llbracket\, A(v_o) = 1 \,\rrbracket = XotX$
0:	$\llbracket\, A(v_o) = 0 \,\rrbracket = XofX$
NOT:	$\llbracket\, A(v_n) = 0 \to A(v_o) = 1 \,\rrbracket, \llbracket\, A(v_n) = 1 \to A(v_o) = 0 \,\rrbracket$
	$= XnDZTofX, XnDZFotX$
AND:	$\llbracket\, A(v_p) = 0 \to A(v_o) = 0 \,\rrbracket, \llbracket\, A(v_n) = 1 \to A(v_m) = 1 \to A(v_o) = 1 \,\rrbracket$
	$= XpDZFofX, XnDZTmDZTotX$
NAND:	$\llbracket\, A(v_p) = 0 \to A(v_o) = 1 \,\rrbracket, \llbracket\, A(v_n) = 1 \to A(v_m) = 1 \to A(v_o) = 0 \,\rrbracket$
	$= XpDZFotX, XnDZTmDZTofX.$
OR:	$\llbracket\, A(v_p) = 1 \to A(v_o) = 1 \,\rrbracket, \llbracket\, A(v_n) = 0 \to A(v_m) = 0 \to A(v_o) = 0 \,\rrbracket$
	$= XpDZTotX, XnDZFmDZFofX.$
NOR:	$\llbracket\, A(v_p) = 1 \to A(v_o) = 0 \,\rrbracket, \llbracket\, A(v_n) = 0 \to A(v_m) = 0 \to A(v_o) = 1 \,\rrbracket$
	$= XpDZTofX, XnDZFmDZFotX.$
XOR:	$\llbracket\, A(v_n) = 1 \to A(v_m) = 1 \to A(v_o) = 0 \,\rrbracket, \llbracket\, A(v_n) = 1 \to A(v_m) = 0 \to A(v_o) = 1 \,\rrbracket,$ $\llbracket\, A(v_n) = 0 \to A(v_m) = 1 \to A(v_o) = 1 \,\rrbracket, \llbracket\, A(v_n) = 0 \to A(v_m) = 0 \to A(v_o) = 0 \,\rrbracket$
	$= XnDZTmDZTofX, XnDZTmDZFotX, XnDZFmDZTotX, XnDZFmDZFofX.$
EQ:	$\llbracket\, A(v_n) = 1 \to A(v_m) = 1 \to A(v_o) = 1 \,\rrbracket, \llbracket\, A(v_n) = 1 \to A(v_m) = 0 \to A(v_o) = 0 \,\rrbracket,$ $\llbracket\, A(v_n) = 0 \to A(v_m) = 1 \to A(v_o) = 0 \,\rrbracket, \llbracket\, A(v_n) = 0 \to A(v_m) = 0 \to A(v_o) = 1 \,\rrbracket$
	$= XnDZTmDZTotX, XnDZTmDZFofX, XnDZFmDZTofX, XnDZFmDZFotX.$
$\to$:	$\llbracket\, A(v_n) = 0 \to A(v_o) = 1 \,\rrbracket, \llbracket\, A(v_n) = 1 \to A(v_m) = 1 \to A(v_o) = 1 \,\rrbracket,$ $\llbracket\, A(v_n) = 1 \to A(v_m) = 0 \to A(v_o) = 0 \,\rrbracket$
	$= XnDZFotX, XnDZTmDZTotX, XnDZTmDZFofX$
$\leftarrow$:	$\llbracket\, A(v_m) = 0 \to A(v_o) = 1 \,\rrbracket, \llbracket\, A(v_n) = 1 \to A(v_m) = 1 \to A(v_o) = 1 \,\rrbracket,$ $\llbracket\, A(v_n) = 0 \to A(v_m) = 1 \to A(v_o) = 0 \,\rrbracket$
	$= XmDZFotX, XnDZTmDZTotX, XnDZFmDZTofX$

Figure 4: Gate words in I' for v_o, depending on $s(v_o)$ (with $o \in \{1, \ldots, N\}$)

$$\begin{array}{ll}
(X1D|ZF4fX, X1t|X) & \vdash_1 (X1tZF4fX, X1DX) \\
(X1D|ZT2DZT4tX, & \vdash_1 (X1tZT2DZT4tX, \\
\quad X1t|X) & \quad X1DX) \\
(X2D|ZF4fX, X2f|X) & \vdash_1 (X2fZF4fX, X2DX) \\
 & \\
(X1tZT|2DZT4tX, X|Y) & \vdash_2 (X2DZT4tX, X1tZTY) \\
(X2fZF|4fX, X|Y) & \vdash_3 (X4fX, X2fZFY) \\
 & \\
(X2D|ZT4tX, X2f|X) & \vdash_1 (X2fZT4tX, X2DX)
\end{array}$$

Splicing systems model recombination, where each reaction takes place as soon as the enzyme and the molecule containing the site are present. Because in a splicing system an unlimited supply of each word is given, even several splicing reactions involving the same word v can occur in parallel. So if we assume that all possible splicing operations occur in parallel, the value of a gate at depth k is computed after at most $4k$ parallel splicing steps: If v_o is a gate whose inputs have already been evaluated, it takes 2 splicing operations to produce a gate value word for v_o from a gate word of the first type, and 4 splicing reactions for a gate word of the second type. The amount of material needed for the splicing system is dependent on the size of the Boolean circuit. For a circuit of N gates, the corresponding splicing system comprises $2N + 2$ rules and up to $4N + 1$ initial words. Interestingly, the splicing rules, the resource that is harder to implement in practice, depend only on the size, not on the form of the Boolean circuit: The Boolean connectives used are expressed solely in the splicing system's initial language.

5 Conclusion

In this paper, we have presented a simulation of Boolean circuits by finite splicing systems. Splicing systems compute in a ”one-pot” reaction without calling for any outside action after the initial ”pouring” of rules and initial words. Our simulation allows, as gate sorts, arbitrary Boolean functions $f : \{0,1\}^i \to \{0,1\}, i \in \{0,1,2\}$. The splicing system computes the value of each gate of the Boolean circuit in time proportional to the gate's depth. The amount of both splicing rules and initial words needed for the simulation is proportional to the circuit size, but the splicing rules are independent of the actual Boolean connectives used, which are described solely in the initial language.

Acknowledgements

Many thanks to Joachim Niehren and Lutz Priese for their important comments on an earlier version of this paper. Furthermore, I would like to thank Harro Wimmel for valuable discussions on splicing systems.

Bibliography

[AD97] Martyn Amos and Paul E. Dunne. DNA simulation of boolean circuits. Technical Report CTAG-97009, Department of Computer Science, University of Liverpool, December 1997.

[CH91] Karel Culik II and Tero Harju. Splicing semigroups of dominoes and DNA. *Discrete Applied Mathematics*, 31:261–277, 1991.

[Erk98] Katrin Erk. Splicing. Master's thesis, FB 4 (Informatik), Universität Koblenz–Landau, Abt. Koblenz, 1998.

[Gat94] R. W. Gatterdam. DNA and twist free splicing systems. In M. Ito and H. Jürgensen, editors, *Words, Languages and Combinatorics II*, pages 170–178. World Scientific Publ. Singapore, 1994.

[Hea87] Tom Head. Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.

[HPP96] Thomas Head, Gheorghe Păun, and Dennis Pixton. Language theory and molecular genetics: Generative mechanisms suggested by DNA recombination. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*. Springer-Verlag, 1996.

[OR96] Mitsunori Ogihara and Animesh Ray. Simulating boolean circuits on a DNA computer. Technical Report TR 631, University of Rochester, Computer Science Department, August 1996.

[Pău96a] Gheorghe Păun. On the splicing operation. *Discrete Applied Mathematics*, 70(1):57–79, September 1996.

[Pău96b] Gheorghe Păun. Regular extended H systems are computationally universal. *Journal of Automata, Languages and Combinatorics*, 1:27–36, 1996.

[Pix95] Dennis Pixton. Linear and circular splicing systems. In *Proceedings of the First International Symposium on Intelligence in Neural and Biological Systems (Herndon, VA)*, pages 181–188. IEEE Computer Society Technical Committee on Pattern Recognition and Machine Intelligence (PAMI), IEEE Computer Society Press, May 1995.

[Str91] Lubert Stryer. *Biochemie*. Spektrum Akademischer Verlag, Heidelberg, Berlin, New York, 1991.