Computing by Observing change in insertion/deletion systems

Peter Leupold\(^{(A)}\)

\(^{(A)}\)Research Group in Mathematical Linguistics
Rovira i Virgili University
Tarragona, Catalonia, Spain
Peter.Leupold@web.de

Abstract
Computing by Observing is a theoretical model for computation that tries to formalize the standard setup of experiments in natural sciences into a model of computation. We explore insertion/deletion systems as underlying systems in a continuation of work on insertion systems. The main result states that insertion and deletion rules with contexts of length one and insertions / deletions of size one suffice to achieve computational completeness in the framework of observing change.

0. Introduction

Much of the recent work in Formal Language Theory has stood in some relation to biochemistry. The original motivation for this were hopes of building actual biocomputers based on the theoretical models that have been developed. Nearly all of these models in the area of DNA computing follow the classical computer science paradigm of processing an input directly to an output, which is the result of the computation. Only the mechanisms of processing are different from conventional models; instead of a finite state control or a programming language it is biomolecular mechanisms that are used, or rather abstractions of such mechanisms.

However, in many experiments in biology and chemistry the setup is fundamentally different. The matter of interest is not some product of the system but rather the change over time observed in certain, selected quantities. To cite two simple examples that might be well-known from High School biology and chemistry classes:

- The predator-prey relationship. It is not of much use to know the numbers of predators and prey in one single moment. The interesting feature here is how the increase or decrease in one of the two populations affects the other population.

\(^{(A)}\)Peter Leupold’s work was done while he was funded as a Beatriu de Pinós researcher by the Departament d’Universitats, Recerca i Societat de la Informació de la Generalitat de Catalunya under contract number 2009 BP-B 00052.
A chemical reaction with a catalyst often has the same product as without, but the energy curves during the reactions are different. In the presence of a catalyst the reaction will occur with less energy consumption and possibly also faster.

The objective of Computing by Observing was to formalize this approach of investigators who have been dealing with biological and chemical systems for ages in a paradigm for computation. The resulting architecture consists of an underlying system, which evolves in discrete steps from one configuration to the next. The second element is an observer, which reads these configurations and transforms them into single letters; a type of classification, if we take the finite number of letters to represent a finite number of classes. In this way, a sequence of configurations is transformed into a simple sequence of symbols, i.e. a string. This corresponds to the protocol of an experiment in biology or chemistry, and for us it is the main result of the computation. Figure 1 depicts this setup, where acceptance is decided based on this observation.

In the initial work on the topic, membrane systems were used as underlying systems [3]. Then plain string-rewriting systems as underlying systems became the main variant [4, 5]; because of their simplicity and generality they seemed suited for identifying key features of the underlying systems that would be crucial for obtaining higher computational power. At the same time, several types of models of bio-computing were investigated as underlying systems — namely splicing systems [2] and sticker systems [1]. The main results of these investigations presented a common pattern: systems that by themselves characterize the context-free languages together with regular observers result in computational completeness.

From the numerous DNA-inspired models of computation [14, 6], here we investigate insertion/deletion systems as underlying systems in the Computing by Observing architecture. Insertion is the operation of inserting a new factor between specified left and right contexts in a string; deletion works analogously, just removing a factor. Motivated from linguistics, systems of insertion rules of this type were first investigated by Galiukschov [7]. Later, more intensive investigations resulted from the fact that insertion and deletion occur in DNA strands. Thus the power of these operations in the field of DNA computing was investigated [9, 10]. An up-to-date and complete overview of work on these systems can be found in the recent dissertation by one of the current authors [11].

In earlier work, the current authors have established that even the simplest possible insertion systems suffice to obtain computational completeness in the Computing by Observing architecture.

**Theorem 1** [12] Every recursively enumerable language can be accepted by an insertion ob-
server system with insertion rules that have empty contexts and insert only one letter.

Therefore also a weaker and somewhat more realistic variant of observation has been considered: observing change. Instead of the entire configuration, the observer only sees what has happened. In our case, this means which rule has been applied. Thus the observation amounts to a sequence of rules as depicted in Figure 2.

For systems with only insertion rules this meant a clear decrease in computing power compared to the observers that read the entire configuration. For attaining completeness, probably insertion rules that have left and right contexts of length two each and insert strings of length two are necessary. Here we investigate in how far the weakening of the observation can be compensated by adding deletion rules to the underlying systems.

Several similar ways of generating formal languages have been studied in Formal Language Theory. Control languages have a grammar’s rules as their alphabet; a derivation is successful only if the sequence of rules that are applied belongs to a certain language [8]. On the other hand, there are Szilard languages [13]. Again the derivation rules form an alphabet, but in this case all derivations are valid. The Szilard language consists of the sequences of rules of possible derivations rather than the terminal words that are generated. The major difference to our approach is that we use the rule sequences for accepting a language rather than for generating it.

1. Observing Insertion/Deletion Systems

We first define insertion/deletion systems and then the way in which we observe them. The reader is assumed familiar with standard concepts from Formal Language Theory such as languages and finite automata; for more details standard textbooks can be consulted [15]. Note that we denote the empty string by $\lambda$.

1.1. Insertion/Deletion Systems

We now formally specify what insertion/deletion systems are and how they implement the insertions and deletions described informally above. Here we do not distinguish terminal and non-terminal symbols for insertion/deletion systems as is often done. Since in our case the systems do not directly generate languages, this distinction would not be meaningful.
Definition 2 An insertion system is a triple \([\Sigma, A, R]\), where \(\Sigma\) is an alphabet, the set of axioms \(A\) is a finite language over \(\Sigma\), and the set of insertion rules \(R\) is a finite set of triples of the form \((u, \alpha, v)\), where \(\alpha \in \Sigma^+\) and \(u\) and \(v\) are either strings over \(\Sigma\) or are equal to \(\Box\) where \(\Box\) is a symbol not in \(\Sigma\).

An insertion rule \([u, \alpha, v] \in R\) indicates that the string \(\alpha\) can be inserted between \(u\) and \(v\); the latter two are therefore called contexts. Stated otherwise, \((u, \alpha, v) \in R\) corresponds to the rewriting rule \(uv \to u\alpha v\). We denote by \(\Rightarrow\) the relation defined by an insertion system. Formally, \(x \Rightarrow y\) iff \(x = x_1uvx_2, y = x_1u\alpha vx_2\), for some \((u, \alpha, v) \in R\) and \(x_1, x_2 \in \Sigma^*\). We denote by \(\Rightarrow^*\) the reflexive and transitive closure of \(\Rightarrow\), and \(\Rightarrow^+\) denote its transitive closure. The language generated by the system is defined as 

\[L := \{w \in \Sigma^* : x \Rightarrow^* w, x \in A\} \]

Now notice that there are two possible meanings of an empty context: on the one hand, it could mean that the context does not matter, and thus any context is admissible; on the other hand, it could mean that there must not be any symbols on the respective side of the insertion site. So far, in insertion systems only the first variant has been considered. It has always been denoted by the empty string, and we will preserve this convention. Thus a rule \((\lambda, a, \lambda)\) says that the letter \(a\) can be inserted anywhere. For the second option, we use the special symbol \(\Box\). Thus a rule \((\Box, a, \lambda)\) says that the letter \(a\) can be inserted at the start of any string, while \((c, bab, \Box)\) says that any string terminating in \(c\) can be prolonged with the factor \(bab\). We see that the use of \(\Box\) allows us to identify the start and the end of the string and thus gives us some more control over the application site of rules.

Definition 3 An deletion system is a triple \([\Sigma, A, R]\), where \(\Sigma\) is an alphabet, the set of axioms \(A\) is a finite language over \(\Sigma\), and the set of insertion rules \(R\) is a finite set of triples of the form \((u, \alpha, v)\), where \(\alpha \in \Sigma^+\) and \(u\) and \(v\) are either strings over \(\Sigma\) or are equal to \(\Box\) where \(\Box\) is a symbol not in \(\Sigma\).

A deletion rule \((u, \alpha, v) \in R\) indicates that the string \(\alpha\) can be deleted between \(u\) and \(v\). The derivation relation and the language specified are defined analogously to insertion systems. We use parentheses to distinguish this type of rules from insertion rules.

For both insertion and deletion systems, we call a system standard if it does not use \(\Box\). Otherwise it is called a system with emptiness check.

Definition 4 An insertion/deletion system is a triple \([\Sigma, A, I, D]\), where \([\Sigma, A, I]\) is an insertion system and \([\Sigma, A, D]\) is a deletion system.

1.2. Insertion/Deletion Change-Observing Acceptors

Now we embed the systems introduced in the preceding definitions into the setup depicted in Figure 2.
Definition 5 An insertion/deletion change-observing acceptor is a quadruple \([\Sigma, \Delta, I, D, F]\), where \(\Sigma\) is the input alphabet, \(I\) is a set of insertion rules over the alphabet \(\Sigma \cup \Delta\), \(D\) is a set of deletion rules over \(\Sigma \cup \Delta\), and \(F\) is a regular language over the alphabet \(I \cup D\) called the decider.

So the set of rules \(I \cup D\) is used as an alphabet for the decider. In practice we will normally assign labels to the rules and use them as the symbols for the observations, i.e. for the sequences of rules used in derivations.

Definition 6 The language accepted by an insertion/deletion change-observing acceptor \(\Omega = [\Sigma, I, D, F]\) is the set of all words \(w \in \Sigma^*\) such that there exists a terminating derivation sequence \(s : w \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n\) with rules from \(I\) and \(D\) such that the corresponding sequence of rules is in \(F\).

The fact that the derivation sequence has to terminate implies that there should not be any insertion rules \([\lambda, ab, \lambda]\) with only empty contexts; these would always be applicable, and consequently there would be no terminating derivation. We illustrate the definition with a simple example.

Example 7 Consider the insertion/deletion change-observing acceptor

\[
\Omega = [(a, b), I, D, (d_1d_2)^*d_3]
\]

with an empty set of insertion rules and the set of deletion rules

\[
D = \{d_1 : (\lambda, a, a), d_2 : (\lambda, b, b), d_3 : (\lambda, ab, \lambda), d_4 : (\lambda, a, \lambda), d_5 : (\lambda, b, \lambda)\},
\]

where for ease of notation of the decider we associate the labels \(d_1\) to \(d_3\) to the rules. The decider guarantees that rules \(d_1\) and \(d_2\) are applied an equal number of times. So the number of deleted \(a\) and \(b\) is equal. The contexts allow only letters to be deleted that are followed by the same letter, i.e. they are in a block of at least two consecutive equal letters. Finally, \(d_3\) guarantees that there is one \(a\) followed by one \(b\). Obviously, words from \(\{a^n b^n : n > 0\}\) can be accepted in this way. If the numbers of occurrences of the two letters are not equal, either \(d_4\) or \(d_5\) will delete the letters left over, and the observation is not in \((d_1d_2)^*d_3\). If the order is different, then not the entire word can be reduced in the way described above. So exactly the language \(\{a^n b^n : n > 0\}\) is accepted.

If we also wanted to accept the empty word, then we would simply have to add the empty string to the decider. Since no rule can be applied on \(\lambda\), it results in an empty observation.

2. Computing by Observing Insertion/Deletion Systems

The basic goal is always to achieve a jump in computational power from the single components to their combination in an observer system. In Theorem [1] three regular components were
sufficient to obtain a computationally complete class of systems. In our current model this is very unlikely. For example, insertion or deletion rules can be applied wherever their application site is found. For two sites $u$ and $v$, the observation of change can in general not distinguish between $uv$ and $vu$; thus any language accepted by an insertion/deletion change-observing acceptor with rules without context will be to some extent commutative in the sequence of factors. However, already with the minimal context size of one we again obtain computational completeness.

**Theorem 8** Every recursively enumerable language can be accepted by an insertion/deletion change-observing acceptor with emptiness check whose insertion and deletion rules both have contexts of size only one, and insert or delete only one letter.

**Proof.** Let $L$ be a recursively enumerable language, and let $M$ be a Turing Machine that accepts it. We can suppose without restriction of generality that $M$ uses empty tape only to the right of the input word. Further, we can suppose that an accepting computation halts in a configuration where the head is on the left-most non-blank tape cell. Now we construct an insertion/deletion change-observing acceptor $\Omega$ that accepts $L$ by simulating $M$.

First $\Omega$ converts the input word into a string that represents the initial configuration of $M$. We use the markers $\$ and $\$$ to delimit the part of the tape that is actually read and rewritten during the computation. The states of $M$ are used to denote the position of the head by putting them to the left of the symbol currently read. If $q_0$ is the initial state, then the initial configuration for an input word $w$ would be represented by a string $\$q_0w\$$ where $\$ represents an empty tape cell and $k$ is an arbitrary natural number. It will be guessed by $\Omega$, and if it is too small to accommodate the computation, then the simulation fails.

The insertion rules that $\Omega$ uses to reach the string $\$q_0w\$$ are $i_l : [q, \$] \rightarrow [p, \$, +]$ for the end markers for all possible tape symbols $x$, $i_r : [\$, x, \$] \rightarrow [\$, $\$, \$]$ for the end markers for all possible tape symbols $x$, $i_0 : [\$, q_0, \$]$ to create the additional working space, and $i_0 : [\$, q_0, \$]$ to introduce the initial state just to the right of the left end marker. This phase is the same for any simulation, thus we require all observations accepted by the decider to start with a prefix from $i_l i_r i_0$. After this we have a string that represents the initial configuration of $M$.

Now we simulate $M$’s computation one step at a time by several insertion and deletion rules. For every transition there is a separate set of rules. First we treat transitions where the head moves to the right. For letters $a$ and $b$ and states $q$ and $p$ of $M$ let

$$t : q, a \rightarrow p, b, +$$

be such a transition. Basically we have to change a factor $qa$ to $bp$. First we delete $a$ by a rule $t_1 : (q, a, \lambda)$ and insert $b$ instead by $t_2 : [q, p, \lambda]$. Now we work in the same way for the left position with rules $t_3 : (\lambda, a, p)$ and $t_4 : [\lambda, b, p]$. Obviously, the corresponding factor that the decider must accept is $t_1t_2t_3t_4$.

A transition to the left is slightly more complicated, because it involves moving the state across a symbol of which we do not know what it is. For such a transition

$$t : q, a \rightarrow p, b, -$$
we first rewrite the letter a by transitions $t_a : (q,a,\lambda)$ and $t_i : [q,b,\lambda]$. Now, for every tape symbol $x$ that can be to the left of $q$ we use a different set of rules for moving and changing the state symbol. By $t_x : [\lambda,x,q]$ we first remove this symbol. Then the state is changed by $t_2 : [\lambda,p,q]$ and the deletion $t_3 : (p,q,\lambda)$. Finally, the symbol $x$ is restored by $t_{x4} : [p,x,\lambda]$.

The corresponding factor that the decider must accept is $t_at_it_xt_2t_3t_{x4}$. Notice how here in some sense the decider language is used as a memory for the letter $x$.

In this way, we can simulate the entire computation of $M$. It remains to detect a final accepting configuration and to make the insertion/deletion system halt. An accepting configuration is characterized by three conditions: the current state is accepting, the head is on the left-most symbol just next to $\diamond$, and no more moves are possible. We first check the last one of these conditions. For every accepting state $q$ and every tape symbol $x$ such that there does not exist a move right of $M$ reading $x$ in state $q$ there is a deletion rule $f_1 : (q,x,\lambda)$; all of these rules have the same label. A further deletion rule $f_2 : (\diamond,q,\lambda)$ removes the state symbol. Now, still the rules $i_0$ and $i_0$ from the initialization would be applicable. For the first one we simply have to remove $\$ by $f_3 : (\lambda,\$,\lambda)$. If we removed $\diamond$ now, then $i_1$ would become applicable again. Therefore we first remove all the other symbols by rules $f_4 : (\diamond,x,\lambda)$ for all possible tape symbols $x$ including $\diamond$. With the rule $f_5 : (\lambda,\diamond,\Box)$ we arrive at the empty string.

The overall decider language is $i_0i_1i_0i_0T^*f_1f_2f_3f_4f_5$ where $T$ is the set of all possible sequences $t_1t_2t_3t_4$ or $t_1t_2t_3t_4t_{x4}$ resulting from the simulation of a transition. If the empty string belongs to $L$ it can be handled as in Example \[7\] \[\square\]

Without the emptiness check we can still attain computational completeness; however, we need to use a few additional rules.

**Corollary 9** Every recursively enumerable language can be accepted by an insertion/deletion change-observing acceptor whose insertion and deletion rules both have contexts of size only one, do not use $\Box$ and insert or delete only one letter.

**Proof.** Basically, a Turing Machine can be simulated as in the proof of Theorem \[8\]. What we lose with the use of $\Box$ is the convenient way of inserting the left and right end markers before starting the simulation. We explain for the left end how we can do without this. The insertion rule this time works without any context, just $[\lambda,\diamond,x]$. This, of course, does not guarantee that the insertion is done at the desired place. For this we add deletion rules $[\lambda,x,\diamond]$ for all the symbols $x$ of $\Sigma$. Such a rule can be applied if and only if $\diamond$ was inserted at a position different from the left end. Thus the application of any rule of this type should lead to immediate rejection by the decider. The right end marker can be treated analogously. \[\square\]

Now the question is, whether we have gained anything in computational power by embedding the insertion/deletion systems into the Computing by Observing architecture. The rules used in the proof of Corollary \[9\] insert and delete always one letter. The context is always of size one, sometimes on the left, sometimes on the right hand side. This kind of system does not seem to have been investigated by itself. The closest comparable result is the one on one-sided insertion/deletion systems that says that $INS_{1,0}^{1,0}DEL_{1,0}^{1,0}$ is a subset of the context-free languages \[11\]. The rules used there are just like ours, only the contexts always have to be on
the same side. It is not clear, if for such systems still computational completeness could be achieved in our context.

If we completely remove the contexts, then there is an immense loss in power, since only commutative languages can be accepted, where commutative means the following: For two letters $x$ and $y$, if a word $u \cdot x \cdot v \cdot y \cdot w$ is in the language, then also $u \cdot y \cdot v \cdot x \cdot w$ is in the language. Thus the order of letters is irrelevant. With longer deletion rules a minimum of order could be imposed like with rule $d_3$ in Example 7. Actually, by changing the rule set there to

$$\{d_1 : (\lambda, aa, \lambda), d_2 : (\lambda, bb, \lambda), d_3 : (\lambda, ab, \lambda), d_4 : (\lambda, a, \lambda), d_5 : (\lambda, b, \lambda)\},$$

we could accept the language $\{a^{2n+1}b^{2n+1} : n > 0\}$ without using any context in the rules. Finding out how big the rules must be for achieving computational completeness without contexts, if this is possible, might be a challenging problem.

3. Outlook

For comparability with earlier results on insertion/deletion systems it would be interesting to see how powerful systems like the ones in Theorem 8 and Corollary 9 are, if the contexts are always on the same side.

Acknowledgments

The author thanks the anonymous referees for their patient and careful reading of a not so carefully prepared first version of this article.

References


