Programming Self-Assembly of DNA Tiles

Marco Bellia and M. Eugenia Occhiuto

Dipartimento di Informatica, Università di Pisa, Italy
{bellia,occhiuto}@di.unipi.it

Abstract. SKI$^{\#}$ is a Turing complete, language for programming in the aTAM model. A compilation technique provides a mapping from SKI$^{\#}$ into finite sets of DNA Tiles that self-assemble in the aTAM model. Though such sets are always finite, the number of Tiles may be relevant, the construction of the Tiles may be heavy and the self-assembly can produce wrong molecular growth. In this paper we discuss the construction of a DNA Universal Machine as an aTAM interpreter for the entire Combinatory Logic, comparing it with the compiler based approach. Finally, Consensus is considered as a case study in distributed programming in the aTAM model and a further step in the design of SKI$^{\#}$ and in the expressivity of aTAM (Wang Tiling) compared to Combinatory Logic and π-calculus.

1 Computing with DNA Tiles

The abstract Tile Assembly Model (aTAM) [1] is (together with its variants) the basic model of DNA Tiles Self-Assembly which is that part of molecular computing where Self-Assembly applies to DNA Tiles. DNA Tiles are molecules of DNA that have a fixed 2D or 3D connection structure and according to this connection structure, these molecules Self-Assemble to form structures which may grow even, indefinitely. Moreover, aTAM is an extension of the Wang formalism of the grid tiling, hence the Tiles are 2D, 4-connection structures that can be (graphically) represented by unit squares, as in Figure 1. Also aTAM Self-Assembly matches the Wang tiling laws (Tiles cover a grid and adjacent Tiles have touching sides of the same color) provided that the bio-chemical stability is satisfied. The equivalence between the Wang formalism and the Turing formalism, as shown in Figure 4, leads to the notions of program and of computation in both aTAM and Wang formalism [1]. A program is a finite set of Tiles. A computation is a grid tiling, i.e. a cover, without holes of the (infinite) plan, using (infinitely) many copies of the Tiles of the program. Figure 3 shows (a part) of a computation where each rectangle represents one sub(program)-computation of the (main) program. Though TM is a milestone in computation, it is not so good for programming because it is lacking of expressivity (for instance, for the composition of functions and of independent computations). For this reason,

[1] extended in aTAM with the actual, bio-chemical structure of Tiles and with the stability properties of the Self-Assembly of such molecules. In aTAM a program is called *system*

other formalisms have been studied for providing languages for programming Self-Assembly of DNA Tiles. Given any computable problem, these languages must provide a development environment for formalizing a solution to the problem and obtaining an aTAM program that Self-Assembles the Tiles according to such a solution. In [2] we considered these languages and proposed the new language SKI$^{\#}$.

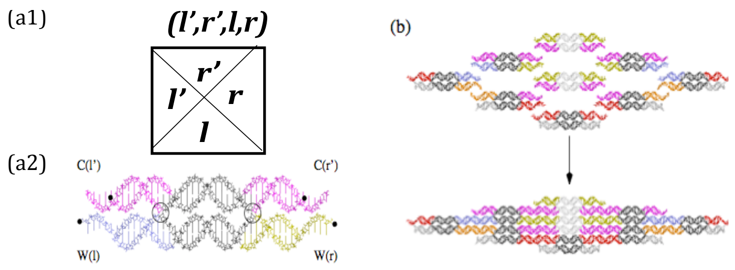


Fig. 1. A (four sticky ends) DNA Tile is a molecule of DNA equipped with four single-stranded, sticky ends which can be used to bind the Tile to other DNA Tiles having sticky ends that are Watson-Crick complementary. (a1) A quadruple of values encoding the sticky ends, and below, a unit square whose sides are labeled (Wang Tiles) by such encodings, provide a linear representation and a graphical 2D representation of a DNA Tile; (a2) A DNA Tile, consisting in a double-crossover molecule in which the sticky ends (indicated by a full dot) are four short sequences of nucleotides GACT, denoted by `C(l')`, `C(r')`, `W(l)`, `W(r)`; (b) Nine DNA Tiles, in the top, that self-assemble in the structure, in the bottom.

2 The calculus SKI$^{\#}$ and its compilation into SKI-Tile

SKI$^{\#}$ [2] is a Turing complete, language designed for programming in the aTAM model and its fundamental characteristic is that it is a proper subset of Combinatory Logic (CL), hence it does not require variable symbols for dealing with bound variables. As a consequence, it provides mechanisms for function definition, application, composition without requiring support for variable scope, binding and substitution. SKI$^{\#}$ is the subset of CL consisting in all the combinatory terms that can be computed using only a finite set of different redexes. Let $\Sigma = S|K|I|\Pi|\mathcal{X}|\Sigma\Sigma$ be the monoid of Combinatory Logic. Let $\mathcal{R} = \{Sabc, Kab, Ia | a, b, c \in \Sigma\}$. A *redex* is any combinator $u \in \mathcal{R}$. The *reductum* of u is denoted by $r(u)$ and is defined by: $r(Sabc) = ac(bc)$, $r(Kab) = a$, $r(Ia) = a$. A *redex-reductum* pair is any pair $(u, r(u))$ for $u \in \mathcal{R}$. A *computation* of $a \in \Sigma$ is any (possibly infinite, sequence) $a_0 \rightarrow_{s_0} \dots \rightarrow_{s_{n-1}} a_n$ where: (1) $a = a_0$; (2) $s_i \in \mathcal{O}(a_i)$; (3) $a_i \downarrow s_i \in \mathcal{R}$; (4) $a_{i+1} = a_i[s_i \leftarrow r(a_i \downarrow s_i)]$. Each computation of a term a is uniquely determined by the indexed sequence of paths to the selected redexes, $\rho = s_0, \dots, s_{n-1}$, and is denoted by ρ or equally, by $a_0 \overset{\rho}{\rightarrow}{}^{*} a_n$. Let $dom(\rho)$ be the range of the indices of ρ and for $i \in dom(\rho)$, $\rho[i]$ be the i-th element of ρ. When the computation is *nonterminating*, ρ is an infinite sequence and $a \overset{\rho}{\rightarrow}{}^{*} \infty$. Given

$a \in \Sigma$, the computation set is $\mathcal{C}(a) \equiv \{\epsilon\} \cup \{\rho_1.\rho_2 \mid a \overset{\rho_1}{\rightarrow}{}^* b \; for \; b \neq a \wedge \rho_2 \in \mathcal{C}(b)\}$, where ϵ is the empty computation such that $b \overset{\epsilon}{\rightarrow}{}^* b$ for all $b \in \Sigma$, and "." is the sequence concatenation. $\mathcal{C}(a)$ is the set of all the computations of a. Let $\rho_a \in \mathcal{C}(a)$ for $a \in \Sigma$. Then $size(\rho_a) = n$ if ρ_a is a finite sequence of length n, $size(\rho_a) = \infty$ otherwise. Moreover, let $U_{\rho_a} \equiv \{a_i \downarrow s_i \mid \rho_a[i] = s_i\}$ be the set of the selected redexes in the computation ρ_a. Then, $rank(\rho_a)$ is the cardinality of U_{ρ_a}. Finally, let $\mathcal{C}^{\#}(a) \equiv \{\rho \in \mathcal{C}(a) \mid rank(\rho) < n \in \mathbb{N}\}$.

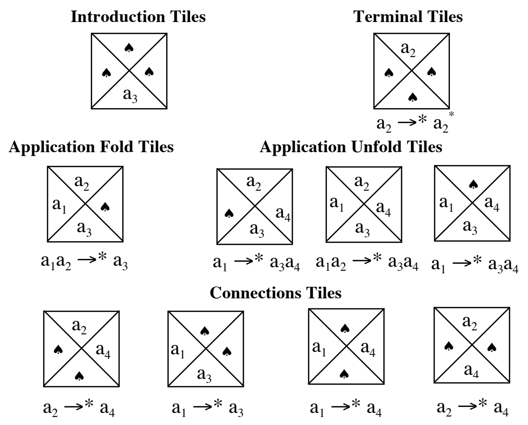


Fig. 2. SKI-Tile is a subclass of Wang Tiles where the colors are combinatory terms: Different terms are different colors, ♠ is a dummy color. The sides of a Tile may be colored with terms that are used (in the program computation) as an application (right or left) part or as a computed value or finally, as "connection" terms for "arranging together" independent parts of the computation. Moreover, the terms in a Tile, must satisfy the law, if any, indicated below the Tile, where $\rightarrow^*$ is transitive closure of combinatory reduction, and $^+$ is marking an irreducible term.

Definition 1 (SKI$^{\#}$). *SKI$^{\#}$ is SKI restricted to the terms*
$$\Sigma^{\#} \equiv \{a \in \Sigma \mid (\exists \rho \in \mathcal{C}^{\#}(a)) \; size(\rho) = \infty \; \vee \; a \overset{\rho}{\rightarrow}{}^* a^*\}$$

Theorem 1 ([2]). *SKI$^{\#}$ is Turing Complete*

Theorem 2 ([2]). *The programs of SKI$^{\#}$ have finite Tile sets of SKI-Tile. Moreover a computable mapping from SKI$^{\#}$ into SKI-Tile exists*

SKI$^{\#}$ leads to the definition of a language of Tile, SKI-Tile, that has the 5 kinds of Tiles of Figure 2, and each kind is designed to support the emulation of combinatory reduction. In [2], we describe a compilation technique from SKI$^{\#}$ program SKI-Tile program and we apply it to the derivation of a finite Tile set for computing applications of the factorial function. The computation (simulation) in aTAM of the resulting SKI-Tile programs leads to computation grids which grow as in Figure 3 and satisfy the grid computation property [2] which states the condition to ensure that independent computations cannot overlap.

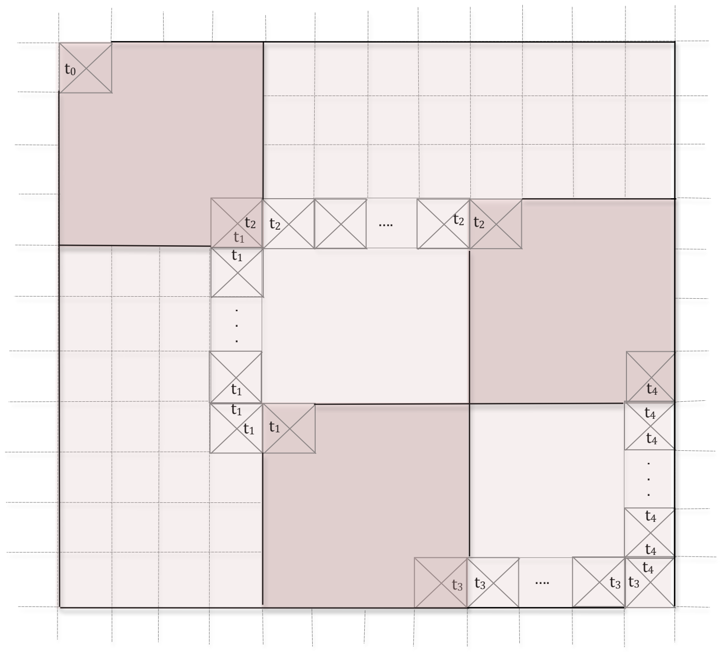


Fig. 3. A Computation grid models the Self-Assembly of a program in SKI-Tile. It grows through the generation of rectangular closed areas whose borders are colored by the dummy value *spadesuit* (in figure represented by a bold line), except for the two Tiles at the top-left and bottom-right corners. Each closed area represents a sub-computation. Areas cannot partially overlap. An area include all the smaller areas that define its sub-computations. In figure, the outermost area is the entire computation corresponding to $t_0 \to^\star t_1 t_2 \to^\star t_3 t_4$. The 3 innermost areas are from left to right, for the computation $t_0 \to^\star t_1 t_2$, $t_1 \to^\star t_3$ and $t_1 \to^\star t_3 t_4$

Roughly speaking, this condition requires that the grid must have each independent computations enclosed in a region delimited by a special dummy color. The kernel of the compilation process is the mapping of each pair $(u, r(u))$, defining a redex-reductum pair of the program, into a Tile of SKI-Tile. The object Tile has sides (resp. sticky ends) colored (resp. configured) by the pair η(u) and η(r(u)) where η is an injection from the terms of SKI$^{\#}$ into colors of Wang Tile (resp. sticky ends of DNA Tiles).

3 A DNA Universal Machine for Combinatory Logic

In this section we describe the definition of a DNA Universal Machine for the programs of the entire Combinatory Logic, including SKI$^{\#}$. The machine consists in a Turing Machine that interprets combinatory programs, and is expressed as an aTAM system (program).

The full machine is too complicated to be presented in this paper, hence we show a simplified version in which the input combinatory term does not contain brackets. Such a simplification allows to understand the fundamentals of the interpretation process, without getting lost in complicated details for dealing with combinators having non atomic arguments. Following [3], a TM is a 7-tuple $\langle Q, \Gamma, \#, \mathcal{I}, \delta, Start, F\rangle$, where Q is the finite set of states, Γ is the finite set of symbols, $\#$ is the blank symbol, $\mathcal{I} = \Gamma \setminus \{\#\}$, $\delta = Q \setminus F \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the function that defines the transition rules that describe the behavior of the machine, $Start \in Q$ is the initial state. $F \subset Q$ is the set of final or halting states. Moreover, we use u^n, u^*, u^w for a sequence of symbols of size n, of arbitrary finite size and for an infinite denumerable sequence of symbols, respectively.

3.1 The Structure of SKI-TM

The machine M we define is such that $\Gamma = \{S, K, I, \} \cup \mathcal{X} \cup \Pi \cup \{\$, \downarrow, \sharp, \triangle_{\mathtt{S}}, \triangle_{\mathtt{K}}, \triangle_{\mathtt{I}} (,)\}$, $F = \{\mathtt{Halt}\}, \delta$ is defined in **Table 2,3,4,5,6**. The other components can be derived from **Table 2,3,4,5,6**, in particular the set of states Q. The initial tape of M is assumed to be constituted of blank symbols $\sharp$ except for a finite sequence. The sequence is supposed to consist of the combinatory program to be reduced, in left associative form, LAF, (without brackets) and delimited by symbol \$ (on both sides). **Table 1** defines the tape syntax. $\mathcal{C}$ are the combinator symbols (we consider only the main combinators), $\mathcal{V}$ are constant and variable symbols, $\mathcal{D}$ are all SKI symbols and $\mathcal{M}$ are markers. Markers include the blank symbol $\sharp$, but do not include \$ since this symbol cannot appear in the sequence[2] Initially the machine is in `Start` and the head is pointing immediately after the leftmost symbol \$.

Table 1 : **Tape Syntax**

$\mathtt{T} ::= \sharp^\omega\ \$\ (\mathcal{D} \mid \mathcal{M})^*\ \$\ \sharp^\omega$	(*Tape*)
$\mathcal{D} ::= \mathcal{C} \mid \mathcal{V}$	(*SKI Symbols*)
$\mathcal{C} ::= \mathbf{S} \mid \mathbf{K} \mid \mathbf{I}$	(*Basic Combinators*)
$\mathcal{V} = \mathcal{X} \mid \Pi$	(*Variable and constant symbols*)
$\mathcal{M} ::= \sharp \mid \downarrow \mid (\mid) \mid \triangle_{\mathtt{S}} \mid \triangle_{\mathtt{K}} \mid \triangle_{\mathtt{I}}$	(*Markers*)

The notation for the rules resorts to the use of set expressions defining a finite set of symbols. A set expression has the form $e_1 \dots e_k$ $(k \geq 1)$, where e_i is either a single symbol of $\mathcal{D} \cup \mathcal{M} \cup \{\$\}$ or the letters `c` or `v`. Each single symbol stands for the singleton containing itself, while `c` stands for the set $\{S, K, I\}$ and `v` for $\mathcal{V}$. Finally, the set expression $e_1 \dots e_k$ stands for the set union of the sets denoted by $e_1 \dots e_k$. Hence rule $(\mathtt{TK}, \mathtt{cv}) \to (\mathtt{TKa}, \mathtt{cv}, \mathtt{R})$ in **Table 3**, is expressing the set of rules $\{(\mathtt{TK}, \mathtt{u}) \to (\mathtt{TKa}, \mathtt{u}, \mathtt{R})\ |\forall \mathtt{u} \in \mathcal{D}\}$. Set expressions are also used in expressing set of states. For instance `TSc:cv` is expressing any state in the set $\{\mathtt{TSu} \mid \mathtt{u} \in \mathcal{D}\}$. As a matter of fact the rule $(\mathtt{TSc{:}cv_1}, \mathtt{cv}() \to (\mathtt{TSc{:}cv_1}, \mathtt{cv}(, \mathtt{L})$, in **Table 5** stands for the set of rules:

$$(\mathtt{TSu_1}, \mathtt{u}) \to (\mathtt{TSu_1}, \mathtt{u}, \mathtt{L})\ \forall \mathtt{u_1} \in \mathcal{D} \wedge \mathtt{u} \in \mathcal{D} \cup \{(\}$$

[2] Round brackets are included since though our assumption on the form of the input program, non atomic arguments are introduced during reduction of combinator **S**.

Table 2 contains the rules for δ relative to the initial state which looks for the leftmost, outermost redex, if any. In effect, the interpreter emulates a leftmost, outermost reduction strategy which is a complete strategy hence, the interpreter always computes the irreducible term, if any. Otherwise, it runs forever.

Table 2 : Looking for redexes

Quintuples looking for a redex	
$(\texttt{Start}, \texttt{v}) \rightarrow (\texttt{Halt}, \texttt{v}, \texttt{L})$	(*No redex found*)
$(\texttt{Start}, \sharp) \rightarrow (\texttt{Start}, \sharp, \texttt{R})$	(*Skip blanks*)
$(\texttt{Start}, \$) \rightarrow (\texttt{Halt}, \$, \texttt{R})$	(*No redex found*)
$(\texttt{Start}, \mathbf{S}) \rightarrow (\texttt{TS}, \triangle_{\texttt{S}}, \texttt{R})$	(*attempt for S*)
$(\texttt{Start}, \mathbf{K}) \rightarrow (\texttt{TK}, \triangle_{\texttt{K}}, \texttt{R})$	(*attempt for K*)
$(\texttt{Start}, \mathbf{I}) \rightarrow (\texttt{TI}, \triangle_{\texttt{I}}, \texttt{R})$	(*attempt for I*)

Tables 3,4 contain the rules to find and reduce redexes with combinator **K** and **I** respectively, which are quite easy and self explaining.

Table 3 : K redex

Quintuples looking for a K *redex*	
$(\texttt{TK}, \$) \rightarrow (\texttt{FF}, \$, \texttt{L})$	(*Attempt fails*)
$(\texttt{TK}, \sharp) \rightarrow (\texttt{TK}, \sharp, \texttt{R})$	(*Skip blanks*)
$(\texttt{TK}, \texttt{cv}) \rightarrow (\texttt{TKa}, \texttt{cv}, \texttt{R})$	(*1st arg has been read*)
$(\texttt{TKa}, \$) \rightarrow (\texttt{FF}, \$, \texttt{L})$	(*Attempt fails*)
$(\texttt{TKa}, \sharp) \rightarrow (\texttt{TKa}, \sharp, \texttt{R})$	(*Skip blanks*)
$(\texttt{TKa}, \texttt{cv}) \rightarrow (\texttt{TKaE}, \sharp, \texttt{L})$	(*Reduction 2nd Atomic arg*)
$(\texttt{TKaE}, \texttt{cv}\sharp)() \rightarrow (\texttt{TKaE}, \texttt{cv}\sharp)(, \texttt{L})$	(*going back to* $\triangle_K$)
$(\texttt{TKaE}, \triangle_{\texttt{K}}) \rightarrow (\texttt{Start}, \sharp, \texttt{R})$	(*Marker* $\triangle_K$ *is deleted*)

Table 4 : I redex

Quintuples looking for an I *redex*	
$(\texttt{TI}, \$) \rightarrow (\texttt{FF}, \$, \texttt{L})$	(*attempt fails*)
$(\texttt{TI}, \sharp) \rightarrow (\texttt{TI}, \sharp, \texttt{R})$	(*Skip blanks*)
$(\texttt{TI}, \texttt{cv}) \rightarrow (\texttt{TIE}, \texttt{cv}, \texttt{L})$	(*arg has been found*)
$(\texttt{TIE}, \sharp) \rightarrow (\texttt{TIE}, \sharp, \texttt{L})$	(*Skip blanks*)
$(\texttt{TIE}, \triangle_{\texttt{I}}) \rightarrow (\texttt{Start}, \sharp, \texttt{R})$	(*Marker* $\triangle_I$ *is deleted*)

Table 5 S **redex**

Blank insertion is omitted	
$(\mathtt{TS}, \mathtt{cv}\sharp) \rightarrow (\mathtt{BKStart}, \mathtt{cv}\sharp, \mathtt{L})$	$(A\ redex\ is\ found)$
$(\mathtt{BKend}, \mathtt{cv}) \rightarrow (\mathtt{TSa}, \mathtt{cv}, \mathtt{R})$	$(First\ argument\ found)$
$(\mathtt{TSa}, \sharp) \rightarrow (\mathtt{TSb}, \downarrow, \mathtt{R})$	$(Insert\ marker\ for\ third\ argument)$
$(\mathtt{TSb}, \mathtt{cv}) \rightarrow (\mathtt{TSb} : \mathtt{cv}, (, \mathtt{R})$	$(Substitute\ second\ argument\ with ()$
$(\mathtt{TSb}{:}\mathtt{cv}, \sharp) \rightarrow (\mathtt{TSbc}, \mathtt{cv}, \mathtt{R})$	$(Rewrite\ second\ argument\ after ()$
$(\mathtt{TSbc}, \mathtt{cv}) \rightarrow (\mathtt{TSbc1}, \mathtt{cv}, \mathtt{R})$	$(Third\ argument\ found)$
$(\mathtt{TSbc1}, \sharp) \rightarrow (\mathtt{TSbc2},), \mathtt{L})$	$(Insert\ )\ after\ third\ argument)$
$(\mathtt{TSbc2}, \mathtt{cv}) \rightarrow (\mathtt{TSc}{:}\mathtt{cv}, \mathtt{cv}, \mathtt{L})$	$(Go\ left\ to\ marker\ for\ the\ third\ argument)$
$(\mathtt{TSc}{:}\mathtt{cv}_1, \mathtt{cv}() \rightarrow (\mathtt{TSc}{:}\mathtt{cv}_1, \mathtt{cv}(, \mathtt{L})$	$(Skip\ second\ argument\ and\ ()$
$(\mathtt{TSc}{:}\mathtt{cv}, \downarrow) \rightarrow (\mathtt{TSS}, \mathtt{cv}, \mathtt{L})$	$(Substitute\ marker\ with\ third\ argument)$
$(\mathtt{TSS}, \mathtt{cv}) \rightarrow (\mathtt{TSS}, \mathtt{cv}, \mathtt{L})$	$(Skip\ first\ argument)$
$(\mathtt{TSS}, \sharp) \rightarrow (\mathtt{BSend}, \sharp, \mathtt{L})$	$(Skip\ blank)$
$(\mathtt{BSend}, \triangle_{\mathtt{S}}) \rightarrow (\mathtt{Start}, \sharp, \mathtt{R})$	$(Delete\ marker\ for\ S\ combinator)$

The rules in **Table 5** deal with the S-redex. In this case, the redex is firstly transformed (by inserting/shifting #) into a more convenient form, then the reduction applies to that form. Let $\sharp^\omega$ \$ u^k $\triangle_S$ $\sharp^i$ a $\sharp^j$ b $\sharp^h$ c $\sharp^m$ u^n \$ $\sharp^\omega$ be a the tape and $\triangle_S$ $\sharp^i$ a $\sharp^j$ b $\sharp^h$ c be the leftmost S-redex to be reduced. The transformation modifies the tape into $\sharp^\omega$ \$ $\tilde{u}^k$ $\triangle_S$ a $\sharp$ b $\sharp$ c $\sharp$ $\tilde{u}^n$ \$ $\sharp^\omega$, where the arguments of the S-redex are separated by exactly 1 blank symbol and $\tilde{u}^k$ and u^k are the same sequence provided that $\sharp$ symbols are ignored. State `BKstart` starts the transformation which ends in state `BKend` (if a S-redex is effectively found. Otherwise, it ends in the state `FF` to deal with failure). Eventually, **Table 5** contains the rules for computing the S-redex of the transformed form and omit the rules of the transformation. The rules in **Table 6** are concerned with the state `FF` that deals with failures in the search of a redex and is responsible for elimination of markers introduced in the attempt of reduction.

Table 6 : Failure of the last attempt

Go backwards for failure	
$(\mathtt{FF}, \mathtt{cv}\sharp) \rightarrow (\mathtt{FF}, \mathtt{cv}\sharp, \mathtt{L})$	$(Skip\ variables,\ constants\ and\ blanks)$
$(\mathtt{FF}, \triangle_{\mathtt{S}}) \rightarrow (\mathtt{Halt}, \mathbf{S}, \mathtt{R})$	$(Substitute\ \triangle_S\ with\ S)$
$(\mathtt{FF}, \triangle_{\mathtt{K}}) \rightarrow (\mathtt{Halt}, \mathbf{K}, \mathtt{R})$	$(Substitute\ \triangle_K\ with\ K)$
$(\mathtt{FF}, \triangle_{\mathtt{I}}) \rightarrow (\mathtt{Halt}, \mathbf{I}, \mathtt{R})$	$(Substitute\ \triangle_I\ with\ I)$

3.2 Compiling SKI-TM into aTAM

To use M, we need to compile it into a program (i.e. a DNA Tile system) for aTAM. It can be accomplished (at least in principle) by using the technique introduced in [4, 5] which provide the stable, bio-chemical conditions for Tile Self-Assembly (salt concentration, temperature thresholds), a robust structure for constructing the Tiles (double, triple crossover) and the right content of the finite Tile set to be used in the Self-Assembly process. Such a content consists in the 4 values of the sticky ends of each Tile of the set. The content is obtained by the mapping introduced in [6], where a TM-rule is compiled into two pairs of Wang Tiles which emulate the rule application on a tape which is represented by a row of the computation grid, where the sequence of rows emulates the sequence of changes in the tape while the machine is running. The mapping is shown in Fig.4 where (a),(b),(c) and (d) are all the possible combinations.

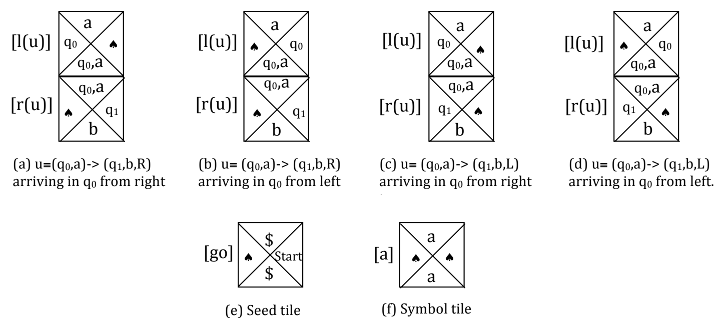

[l(u)]
[r(u)]
(a) u≡(q0,a)-> (q1,b,R) arriving in q0 from right
(b) u≡ (q0,a)-> (q1,b,R) arriving in q0 from left
(c) u≡ (q0,a)-> (q1,b,L) arriving in q0 from right
(d) u≡ (q0,a)-> (q1,b,L) arriving in q0 from left.
[go]
Start
[a]
(e) Seed tile
(f) Symbol tile

Fig. 4. Wang's encoding of rules with Tiles, where $\mathtt{a} \in \Gamma, \mathtt{q_i} \in \mathtt{Q}$. In addition to tape and state symbols, tile colors contain a distinct color $q.a$ for each pair (q, a), head of a rule $\mathtt{u} \in \delta$. (a)-(d) Each rule is encoded with two distinct tile pairs: The use of one or of the other pair depends on the (left or right) side on which the simulation of the MDT head shift is arriving. The two tiles, in each pair, are labeled [l(u)], resp. [r(u)] since encode the left, resp. right, part of the rule u. (e) The initial state and head position is encoded by the tile labeled [go] which is part of the seed of each computation grid. (f) Each tape symbol a, including #, is encoded by a tile labeled [a] as in the figure. These tiles allow to form the tiles representing the combinatory program to be evaluated, they are part of the seed and may be reproduced inside the grid, when needed.

Fig.5 shows the computation grid resulting from the use of M in the evaluation of a combinatory program that begins with the sequence KIS. The first row represents the input contained initially in the tape. The seed tile appears under the invariant tile of the leftmost \$, representing the initial position. The sequence of row contains the Tiles that emulate the machine M that reduces the combinatory term.

...	#	\$	[K]	[I]	[S]	....	\$	#	...	
	#	[go]	[l(u1)]	[I]	[S]	....	\$	#		
	#	\$	[r(u1)]	[l(u2)]	[S]	....	\$	#		u1≡(Start,K)->(TK, Δ_K, R)
	#	\$	[Δ_K]	[r(u2)]	l(u3)	....	\$	#		u2≡ (TK,I)->(TKa, I, R)
	#	\$	[Δ_K]	[l(u4)]	r(u3)	....	\$	#		u3≡ (TKa,S)->(TKaE, #, L)
	#	\$	[l(u5)]	[r(u4)]	#	....	\$	#		u4≡ (TKaE,I)->(TKaE, I, L)
	#	\$	[r(u5)]	....	#	....	\$	#		u5≡ (TKaE, Δ_K)->(Start, #, R)
...	...	....	....	....	....	....	\$	...	...	

Fig. 5. Each element containing a symbol in Γ, represents an invariant tile with that symbol. The other elements represent tiles for the left or right part of the rule which is applied. The rules applied are shown at the end of the row containing the tile encoding the right part of the rule.

We conclude this section noting that a different way exists for implementing SKI-TM in aTAM. It consists in compiling a Universal Turing Machine and then provide for the encoding of SKI-TM into the program representation of such a machine. This way is particularly interesting in view of [7].

3.3 Pros and Cons of M compared with SKI# and its compiler

Even if the compiler for SKI$^{\#}$ has some remarkable aspects, its use has also, some practical limitations which include:

- Deterministic growth. Since M uses a specific (complete) reduction strategy, the grid grows deterministically using M. The same does not hold for the use of the compiler. Even limiting the set of the redex-reductum pairs to those of the outermost redexes we may have a pair that applies to a non-outermost redex of an intermediate reduced term;
- Construction in aTAM, of all the Tiles in the object finite set. M requires always the same set of Tiles. In SKI$^{\#}$, the number of different Tiles depends on the specific program.
- Reuse and/or modification of programs. M requires always to change the Tiles of the first row of the grid. SKI$^{\#}$ programs are function applications hence when the input changes the program must be re-compiled. However the new Tile set may have a relevant set intersection with the previous application.
- Errors due to the violation of the grid property. The computation grids of M do not need grid property.
- Control of the shape of the computation grid. The computation grid, in M, is very far from the source program and contains symbols of the emulator. In the computation grids of SKI# the Tiles contain only terms of the source programs.
- Expressivity. M has the expressivity of CL, whilst SKI$^{\#}$ of a subset of CL.

4 Consensus in DNA Tile Self-Assembly

Starting from the definition of Consensus in DNA Tile Self-Assembly, we consider distributed computation, extend our approach mixing SKI calculus with process algebras, introduce an alternative compilation of SKI$^{\#}$ into SKI-Tile. We introduce and discuss it through the example of the algorithm of consensus [8]. We show how it could be expressed in the extended SKI calculus and then, compiled into an aTAM system.

In distributed computing, Consensus is the well known problem in which a fixed number of agents require to agree on 1 among a finite set S of values. In the version of [8, 9], it consists in a 3-state (one-way) population protocol in which S={p,n,u} is the agent state set and agents cannot crash. Agents can communicate in pairs for letting know the current value of their own state and possibly, changing it according to the protocol rules. The rules state that two communicating agents having: (1) same state, maintain such a state; (2) one state p (for positive opinion), the other state n (for negative o.), both pass in state u (undecided o.); (3) one state p (resp. N), the other state u, both pass in state p (resp. N). We give below, two algebraic formulations of the protocol using Milner's π-calculus [10].

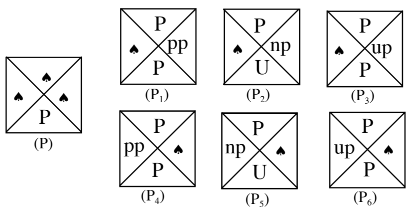
$$\begin{aligned}
&\mathtt{P(c)} := \overline{\mathtt{c}}\mathtt{p.c(x).P'(c,x)} + \mathtt{c(x)}.\overline{\mathtt{c}}\mathtt{p.P'(c,x)}\\
&\qquad \mathtt{P'(c,x)} := [\mathtt{x} = \mathtt{p}]\mathtt{P(c)} + [\mathtt{x} = \mathtt{n}]\mathtt{U(c)} + [\mathtt{x} = \mathtt{u}]\mathtt{P(c)}\\
&\mathtt{N(c)} := \overline{\mathtt{c}}\mathtt{n.c(x).N'(c,x)} + \mathtt{c(x)}.\overline{\mathtt{c}}\mathtt{n.N'(c,x)}\\
&\qquad \mathtt{N'(c,x)} := [\mathtt{x} = \mathtt{p}]\mathtt{U(c)} + [\mathtt{x} = \mathtt{n}]\mathtt{N(c)} + [\mathtt{x} = \mathtt{u}]\mathtt{N(c)}\\
&\mathtt{U(c)} := \overline{\mathtt{c}}\mathtt{u.c(x).U'(c,x)} + \mathtt{c(x)}.\overline{\mathtt{c}}\mathtt{u.U'(c,x)}\\
&\qquad \mathtt{U'(c,x)} := [\mathtt{x} = \mathtt{p}]\mathtt{P(c)} + [\mathtt{x} = \mathtt{n}]\mathtt{N(c)} + [\mathtt{x} = \mathtt{u}]\mathtt{U(c)}\\
&\mathtt{A(c)} := \mathtt{P(c)} \mid \ldots \mid \mathtt{P(c)} \mid \mathtt{N(c)} \mid \ldots \mid \mathtt{N(c)} \mid \mathtt{U(c)} \mid \ldots \mid \mathtt{U(c)}
\end{aligned}$$

where, capital case symbols $\{\mathtt{P}, \mathtt{P'}, \mathtt{N}, \mathtt{N'}, \mathtt{U}, \mathtt{U'}\}$ are names for process definitions, all lower case symbols $\{\mathtt{p}, \mathtt{n}, \mathtt{u}\}$ are names for process links/channels, finally `x` is the only variable symbol. `P+Q` is the choice operator, `P|Q` is distributed operator, [x=y]P is match operator, P:=Q is recursive process definition. Since it uses a finite set of values, a re-formulation of A(c) in the system B below, can be obtained. System B consists in a syntactic transliteration of A(c) which introduces 3 distinct channels for sending and receiving, and can execute without resorting to the variable substitution mechanism.

$$\begin{aligned}
&\mathtt{P} := \overline{\mathtt{p}}\mathtt{p.P} + \overline{\mathtt{n}}\mathtt{p.U} + \overline{\mathtt{u}}\mathtt{p.P} + \mathtt{p(x).P} + \mathtt{n(x)U} + \mathtt{u(x)P}\\
&\mathtt{N} := \overline{\mathtt{p}}\mathtt{n.U} + \overline{\mathtt{n}}\mathtt{n.N} + \overline{\mathtt{u}}\mathtt{n.N} + \mathtt{p(x).U} + \mathtt{n(x)N} + \mathtt{u(x)N}\\
&\mathtt{U} := \overline{\mathtt{p}}\mathtt{u.P} + \overline{\mathtt{n}}\mathtt{u.N} + \overline{\mathtt{u}}\mathtt{u.U} + \mathtt{p(x).P} + \mathtt{n(x)N} + \mathtt{u(x)U}\\
&\mathtt{B} := \mathtt{P} \mid \ldots \mid \mathtt{P} \mid \mathtt{N} \mid \ldots \mid \mathtt{N} \mid \mathtt{U} \mid \ldots \mid \mathtt{U}
\end{aligned}$$

The use of distinct channels allows an instantaneous state transition of both the agents at each communication of an agent pair (the formulation B required two distinct communications in sequence). Figure 6 contains a first (uncompleted) formulation in DNA Tile of system B. Each process is expressed by 6 different Tiles, one for each of the 6 terms occurring in the +_expression defining a process in system B. In Figure 6, the east side of the tile is assumed to be used for receiving whilst the west side for sending. This results into an asymmetry in the communication of the agent pairs (which is not present in the latter π-formulation), in addition the computation grid of the aTAM model is too rigid in "selecting" the tiles to be considered for the grid growth (in contrast to the terms of a |_expression which can be coupled in any way). Figure 7 contains the additional Tiles that a process requires to get in contact with any process in the system and arbitrarily positioned in a column of the computation grid.

Proving equivalence between the π-calculus formulation of system B and its formulation in aTAM is out of the scope of the paper. However, we have shown lines along which we can pass from algebraic formulations of some distributed systems to the DNA Tiles for the execution of such systems by self-assembly in



P := p*p.P + n*p.P + u*p.U + p(x).P + n(x).U + u(x).P
N := p*n.U + n*n.N + u*n.N + p(x).U + n(x).N + u(x).N
U := p*u.P + n*u.N + u*u.U + p(x).P + n(x).N + u(x).U

B = P | ... | P | N | ... | N | U | ... | U

Fig. 6. P, N, U are the positive, negative and undecided kind of agent resp. The tiles for an agent of kind P are in the top (those for the other two, look similarly). Each tile has the current state of the agent in the north side, the current communication channel either in the east or in west side, the action to do in the south side. An algebraic formulation of consensus system is given in the bottom, in π-calculus: One row for each kind of agent (for typographical reasons an output channel p is here, denoted by p^* instead of $\bar{p}$). The last row is the full system and consists of a fixed number of agents of each of the three kinds: Agents can interact in pairs in any oder, provided that the corresponding tiles have the communication side of the same name/color.

the aTAM model. Figure 7 shows on its right part, the behavior of 3 contiguous agents in a computation grid of the consensus system.

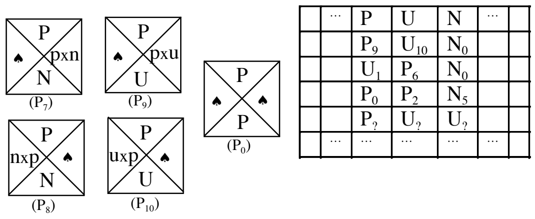


Fig. 7. A possible behavior of three agents of the Consensus protocol in the aTAM model. The π-formulation of system B, given in Fig.6, is for distributed (possibly concurrent) computation. To run in the sequential aTAM model, each agent must be enriched by a stand-by behavior in which the agent is not communicating. This behavior is expressed, for agents of kind P, by the Tile labelled P_0 (where both east and west sides are ♠). In addition, to guarantee that agents can communicate in arbitrary pairs, each agent must be able to exchange its position with a contiguous agent, in a row of the grid. This is allowed, for agents of kind P, by the four Tiles that are labelled P_7-P_{10}. The grid, in the right part of the figure, shows the behavior of the 3 contiguous agents, in the first row. In the second row, the first and the second agent exchange the respective positions and then communicate (in the third row). As a consequence, the undecided agent, in first column, becomes a positive agent and enters in stand-by in the next step (shown in the next row) whilst the other two agents, one positive and one negative, communicate in between and both become, in the last row, undecided agents.

We conclude the paragraph, recalling that concurrent computation is a mix of distributed and parallel computation. Since the structure of programs (namely, finite sets of DNA Tiles) and the properties of the basic computation mechanism (namely, Tile self-assembly in a grid), the aTAM model copes with distributed computation but it is a sequential computation model. This limitation is the result of various considerations on some bio-chemical aspects of DNA molecular interactions [11, 4]. Hence, the definition of a parallel, computation model for DNA Tile self-assembly should reconsider such aspects. This is an interesting line of investigations to allow concurrent computations in DNA Tile self-assembly.

5 Conclusions

The paper introduced an aTAM Universal Machine for CL. Apart from the interest in itself, its use is compared with the one of the language SKI$^{\#}$ and of its interpreter [2]. SKI$^{\#}$ is a Turing complete, language designed for programming in the aTAM model and is a proper subset of CL. The comparison shows that the use of the Universal Machine appears more convenient than the use of SKI and its compiler, when the number of the required Tiles is considered. Moreover the use of the Universal Machine does not require any checking for the grid property. The paper discusses a formalization of the Consensus protocol in the aTAM model. It should be interesting to compare it with the one given in [9] for Strand Displacement Systems. Finally we note that the given formalization was obtained by simplifying a previous one given in π-calculus and using variables. The simplification was based on the removal of the variables that were ranging on finite sets of values. This can lead to an integration of SKI$^{\#}$ with operators of a process algebra with variables ranging on finite sets of values.

References

1. Rothemund, P.W.K., Winfree, E.: The Program Size Complexity of Self-Assembled Squares - [revised may 20 - 2000]. In: ACM Symposium on Theory of Computing (as Extended Abstract). (2000) 459–468
2. Belia, M., Occhiuto, M.E.: DNA Tiles, Wang Tiles and Combinators. In: Proc. of CS&P'2013. CEUR vol.1032 (2013) 114
3. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. 2nd edn. AddisonWesley, Higher Education (2003)
4. Winfree, E.: Simulations of Computing by Self-Assembly. In: 4th DIMACS Meeting on DNA Based Computer. (June 1998)
5. Winfree, E., Liu, F., Wenzler, L.A., Seeman, N.C.: Design and Self-Assembly of Two-Dimensional DNA Crystals. Nature **394** (1998) 539–544
6. Wang, H.: Dominoes and the AEA case of the Decision Problem. In: Symp. on Mathematical Theory of Automata. (1963) 23–55
7. Neary, T., Woods, D.: Four Small Universal Turing Mchines. In: Proc. of 5th. MCU. LNCS 4664 (2007) 242–254
8. Angluin, D., Aspnes, J., Eisenstat, D.: A simple population protocol for fast robust approximate majority. Distributed Computing **21** (2008) 87–102
9. Chen, Y.J., et al: Programmable chemical controllers made from dna. PNAS **97**(3) (2000) 984–989
10. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, I-II. Information and Computation **100** (1992) 1–77
11. Winfree, E., Yang, X., Seeman, N.: Universal Computation via Self-Assembly of DNA: Some Theory and Experiments. In: 2th DIMACS Meeting on DNA Based Computers. (June 1996)