Int. J. Communications, Network and System Sciences, 2013, 6, 260-276
http://dx.doi.org/10.4236/ijcns.2013.65029 Published Online May 2013 (http://www.scirp.org/journal/ijcns)

All-in-One: Space-Time Body, Function and Metric — A Fundamentally New Approach to Computation

Hermann von Issendorff
Institut für Netzwerkprogrammierung, Hemmoor, Germany
Email: hv@issendorff.de

Received February 21, 2013; revised April 21, 2013; accepted May 9, 2013

Abstract

Every algorithm which can be executed on a computer can at least in principle be realized in hardware, *i.e.* by a discrete physical system. The problem is that up to now there is no programming language by which physical systems can constructively be described. Such tool, however, is essential for the compact description and automatic production of complex systems. This paper introduces a programming language, called Akton-Algebra, which provides the foundation for the complete description of discrete physical systems. The approach originates from the finding that every discrete physical system reduces to a spatiotemporal topological network of nodes, if the functional and metric properties are deleted. A next finding is that there exists a homeomorphism between the topological network and a sequence of symbols representing a program by which the original nodal network can be reconstructed. Providing Akton-Algebra with functionality turns it into a flow-controlled general data processing language, which by introducing clock control and addressing can be further transformed into a classical programming language. Providing Akton-Algebra with metrics, *i.e.* the shape and size of the components, turns it into a novel hardware system construction language.

Keywords: Topological space, Dataflow Machine, Many-Sorted Logic, DNA Programming, Layout Simplification

1. Introduction

Living nature demonstrates how to program discrete spatiotemporal systems: It generates chains of amino acids from genetic code [1]. In a suitable environment the chain of amino acids then folds to a protein, *i.e.* a one-dimensional formation mutates into a three-dimensional one. The transaction can be reversed by changing the environment. This shows that there is a bijective and bicontinuous mapping between the chain of amino acids and the protein, called homeomorphism [2]. With other words: The chain of amino acids represents a program which contains the complete spatial information of the protein. The spatial structure of the protein arises from additional weaker binding forces between the amino acids. By external impact, e.g. by adsorption of another molecule, the structure of the protein may change into another metastable state. With other words: A protein has the ability to store and process information.

Discrete physical systems usually are a composition of a set of three-dimensional material components or any abstraction thereof. If the components are active, *i.e.* if they produce physical objects or evaluate functions, then they are temporally directed and are activated in a partial temporal order. If the components are static, then they can be assigned a partial assembling order which also induces a temporal direction.

Abstracting a discrete physical system, for instance a computer, from its metrics, *i.e.* from the spatial measures of its components, the residue is a three-dimensional directed network of the executable functions which are realized by the components.

Abstracting a discrete system, for instance again a computer, from its functionality, the residue is a three-dimensional directed network of building blocks which have the size and shape of the system components. Abstracting a discrete system from the metrics and the functionality at the same time, the residue is a directed network of nodes showing their dependencies. There, any two nodes may be related or not, and each node may be related to any finite number of preceding and succeeding nodes.

If the nodes of the network are provided again with their original concrete functional and metric properties, then the original system is regained. The nodal network is therefore a structural class of all discrete physical systems. Thus, if there is a formal language for the spatiotemporal description of a directed relational nodal net-

work, it is a common language for all discrete physical systems. This even includes every kind of computer, no matter, if it applies to a von-Neumann architecture or not. Akton-Algebra, for short *AA*, provides this capability.

It is important to notice that abstraction from metrics and functionality does not mean abstraction from space and time. The latter would reduce a directed discrete system to a directed graph, *i.e.* to a mathematical object of graph theory, which does not have any relation to space or time. The spatial relations between the nodes, however, are the very properties on which *AA* is based upon. An *AA*-node, *i.e.* the metric abstraction of a component, does have an arbitrary but non-zero shape and size, and traversing the node from its front-end to its back-end takes an arbitrary but non-zero amount of time. This means that an action of a component does not disappear under metric and functional abstraction but is only reduced to a rudimentary action of propagation. This is the reason why the word "*Akton*" has been chosen as a general designator of a concrete component and any of its metrical or functional abstractions.

A spatial description of spatiotemporal structures by programming has not been considered up to now. Classical data processing languages are not provided with spatial semantics. They do not need to because they are tailored to the sequential execution of the von-Neumann-computer. At first glance, the graphical calculus proposed by [3] seems to have some similarity to *AA*. Their calculus, however, is aimed at quantum informatics and does not describe classical physical structures. Thus, the only paper on spatiotemporal structures seems to be an early one by the author himself [4].

The paper proceeds as follows: In the next section the fundamental elements of a nodal network are analyzed. A topological cut needs to be introduced in order to resolve crossings in spatial structures, and a second cut to resolve cycles and crosslinks in planar structures. This gives rise to a hierarchy of *Akton sorts* and a hierarchy of *Interface sorts*. The language of abstract *AA* is then synthesized from these two fundamental hierarchies. There are four sets of production rules, which stepwise generate a programming language of increasing power. The first *set* of production rules introduces an *AA* language for the abstract description of planar and antiparallel structures, the second *set* extends the *AA* language to represent symbolic networks, the third one extends it to describe digital or analog functional structures, and the fourth one to even comprise metric structures.

The second part of the paper is devoted to symbolic, functional and metrical concretizations. The application of the symbolic language specification already suffices to describe spatial structures of considerable complexity. Surprisingly, the language of antiparallel linear structures generates four fundamental substructures which may be interpreted as the four genetic instructions of life, chemically known as the nucleotides guanine, adenine, thymine, and cytosine.

Introducing functionality into *AA* requires the extension of both the *Akton* and the *Interface* hierarchies. This is achieved by introducing digital values as subsorts of the *Interface* hierarchy. Although not elaborated in detail, analog instead of digital values could be introduced as well. Finally, in order to expand *AA* into a metric language requires the extension of *Akton sorts* and *Interface sorts*. For this purpose, several basic geometrical structures need to be introduced as for instance *multiple links*, *multiple forks*, *multiple joins* as well as topological cuts. This way *AA* always permits the constructive representation of every digital or analog electronic circuit on two layers only. These features considerably simplify the layout of highly integrated electronic circuits.

The conclusions finally subsume the essential achievements of *AA*.

2. Elements and Elementary Structures of AA

As observed in the introduction, an abstract nodal network is a common structure underlying every discrete physical system. In general, a nodal network of a discrete physical system has a three-dimensional structure. A formal description, on the other hand, is an ordered sequence of symbols and thus has a one-dimensional structure. The aim of this chapter is to show that a three-dimensional nodal network can be mapped into a one-dimensional description and vice versa, without losing any structural information.

The class of nodal networks we are dealing with is always assumed to be directed. If the physical system underlying the network is not directed or does not have an entry and an exit, these features can be introduced without the loss of generality. Each node of a directed nodal network has two interfaces, one for the input and one for the output.

The representation of a directed abstract nodal network can be done at different levels of detail. As depicted by the hierarchy in **Figure 1**, new levels of *sorts* with more and more properties can be added. At the top level, the network is represented by the general *sort Akton*. At the second level, called the *fundamental* level, there are four *sorts* of nodes called *Head*, *Body*, *Tail* and *CS* (*Closed System*). The relations between the *sorts* are specified by means of interfaces. At the *fundamental* level, there are only two primitive *sorts* of interfaces, the non-empty one (symbolized by $\neg\varepsilon$) and the empty one (symbolized by ε). *Sort Head* has no preceding nodes, *i.e.* an empty *input* but at least one succeeding node, *i.e.* a non-empty *output*. *Sort Tail* has no succeeding nodes but at least one preceding node, *i.e.* a non-empty *input* and an empty *output*. *Sort Body* has at least one preceding

node and at least one succeeding node, *i.e.* a non-empty *input* and a non-empty *output*.

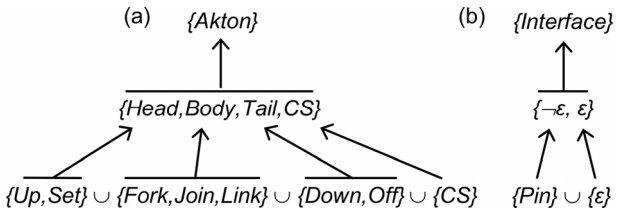


Figure 1. (a) The hierarchy of abstract *Akton sorts*. These three levels are common to all discrete systems. The third level comprises the set of basic abstract *sorts*. (b) The hierarchy of *Interface sorts* is depicted at the right. At the third level, the non-empty *Interface sort* is quantified by a basic element called *Pin*.

Finally, sort *CS* has no preceding and no succeeding nodes, *i.e.* input and output are empty.

The third hierarchical level, called abstract structural level, consists of *basic abstract sorts*. Their non-empty *inputs* and *outputs* are specified by a quantified basic *Interface* element called *Pin*. There are three basic *subsorts* of *Body* called *Fork*, *Join* and *Link*. Fork has one *Pin* in the *input* and two *Pins* in the *output*, *Join* has two *Pins* in the *input* and one *Pin* in the *output*, and *Link* has one *Pin* in the *input* and one in the *output*. The basic *subsorts* of *Head* are called *Up* and *Set* each of them having a single *Pin* in the *output*. The *basic subsorts* of *Tail* are called *Down* and *Off*, each having a single *Pin* in the *input*.

The subsorts of *Head* and *Tail* differ by their semantics. *Up* and *Down* as well as *Set* and *Off* are necessary for mapping the three dimensions of nodal networks to the single dimension of a string of symbols, *i.e.* the program code. This mapping needs to be explained more closely.

Because of the abstraction from metrics the structure of a nodal network is a topological one. A topological structure preserves the adjacency of the nodes, if it is mapped into another shape by a function called homeomorphism. Even cuts are admissible as long as the correlation between the cutting ends is guaranteed. Homeomorphism is bijective and bicontinuous. This means that an original topological structure may arbitrarily be distorted but can always be regained by reversing the homeomorphism.

In order to describe the topological properties of the nodal network, a topological frame of reference is needed. Since there is no metrics, the frame of reference can only be relational. Such frame of reference can be defined by referring to a human observer who physically differentiates between three independent pairs of inverse spatial relations, *i.e. left-right*, *above-below* and *front-back*.

In addition, a privileged direction in the frame of reference needs to be selected in order to orient the directed nodal system, *i.e.* on which side to place the elements of *sort Head* and on which side the elements of *sort Tail*. Following the reading standard of the Western Hemisphere, the direction from *left* to *right* is chosen. Since every node represents an action and action takes time, the orientation also introduces a direction of time. Thus *left* will also be interpreted as *earlier* and *right* as *later*.

1) The mapping of the nodal network from a three-dimensional representation to the one-dimensional description of *AA* requires several steps: The nodal network has to be oriented so that all system *Entries* are at the left side and all system *Exits* are at the right side.

2) The nodal network has to be projected to an oriented plane of observation spanned between the *left-right axis* and the *above-below axis* and positioned between the nodal network and the observer. Usually, this projection will give rise to crossings of nodal connections (see **Figure 2**). Since the crossings are spatial residues they must be removed. This is achieved by cutting the rear connection and replacing the cutting ends by a pair of *Down* and *Up*, *Down* being a *subsort* of *Tail* and *Up* being a *subsort* of *Head* as mentioned before.

3) The resulting planar network may still contain non-orientable structures like cycles or crosslinks (see **Figure 3**). This problem can be solved in a way similar to the crossing problem by cutting the structures and inserting a pair *Off* and *Set,* representing a cut in the plane. *Off* is a *subsort* of *Tail* and *Set* is a *subsort* of *Head* as mentioned before.

Figure 2. Planarization of a spatial structure, *i.e.* the removal of a crossing, is achieved by cutting the rear connection and inserting a pair of *Aktons* instead called *Down* and *Up*.

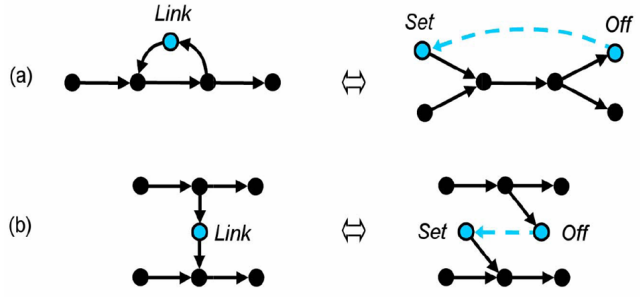


Figure 3. Orientation of cycles (a) and crosslinks (b) from *left* to *right* by cutting and inserting a pair of *Aktons* instead called *Off* and *Set*.

4) The separate utilization of spatial and planar cuts does not suffice to linearize every spatial nodal network. There are nodal structures where both cuts are to be applied crosswise. These structures can be characterized by two antisense strands which are interconnected by several crosslinks. A simplified structure of this kind is depicted in **Figure 4** showing two antisense strands which are connected by two crosslinks. The directions of these crosslinks are not specified on purpose. An orientation of them can be accomplished by either *left-* or *right-twisting* the contrarily oriented strand thus generating different crossings of the crosslinks.

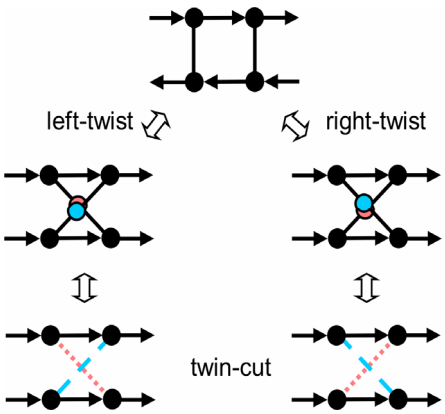


Figure 4. Simplified structure of two antisense strands connected by two undirected crosslinks.

The planarization of the crossings is achieved by applying a spatial cut to the rear crosslink as shown by the dotted red line, and a planar cut to the upper one as shown by the dashed blue line. Finally assigning directions to the crosslinks amounts to four different twin-cuts for the *left-twisted* as well as the *right-twisted* structure. The twin-cuts of the left-twisted structures are depicted in detail in **Figure 5**.

A nodal network can now formally be represented by a string of symbols, *i.e.* in linear form. To this end, two adjacent independent subnetworks *x* and *y* are related by an infix symbol "/", called *Juxta*, where *x/y* means *x* lies *above y*. Likewise, two adjacent dependent subnetworks *x* and *y* are related by an infix symbol ">", called *Next,* where *x>y* means *x precedes y* in space and time. In order to reduce the amount of parentheses *Juxta* is assumed to bind stronger than *Next*.

3. Definition of Abstract AA

In the previous chapter we discovered a way how to turn a three-dimensional network of nodes into a linear oriented network of a nodal string, *i.e.* a string of abstract *Aktons*. A nodal string can be read and processed like a program, which under abstraction from functionality and metrics means to reconstruct the original spatial topological structure of the nodal network. Thus we already know that there exists a formal language for the description of nodal networks. In particular, we discovered the set of basic structural nodes of *AA*. However, what we cannot do up to now is to synthesize a nodal system, because we do not know the rules by which nodal subnetworks are to be related. In formal language terms, we need to know the grammar of *AA*. That is what we will do in this chapter.

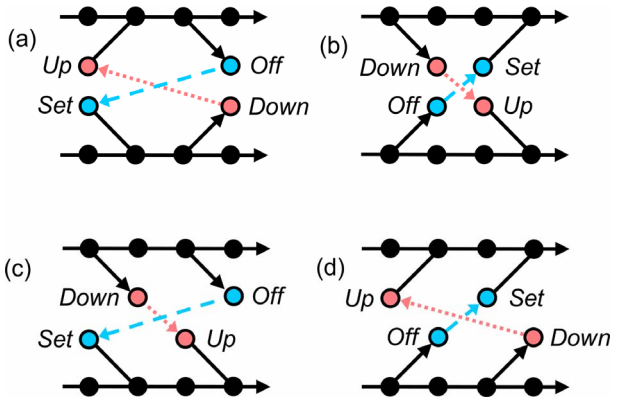


Figure 5. Detailed view of four left-twisted twin-cut structures (a), (b), (c), (d) which can be derived from the undirected twin-cut structure of figure 4.

AA is a *many-sorted term-algebra*, throughout defined by first order logic. It is built up from a hierarchy of *Akton sorts* and a hierarchy of *Akton Interfaces*. The hierarchies can be formally described by a general function *abstract* which maps each lower level set of *subsorts* to its *immediate upper sort*. The grammar of *abstract AA* is systematically derived descending the two hierarchies of *sorts* as shown in **Figure 1**. There, the first hierarchy is headed by *sort Akton*, the other one by *sort Interface*. These general notions will step by step be specified by adding more properties while descending the hierarchical levels further down.

Aktons and the relations *Next* and *Juxta* are destined to describe directed nodal networks. Adjacent directed *Akton terms* may either be dependent or independent. Dependency is expressed by the relation *Next* and independency by the relation *Juxta*. *Next* and *Juxta* produce a free semigroup FA^+ of *fundamental Akton terms*.

The *Akton term variables* defined here and further down by means of tables are represented by using the proper names of the *sets of term sorts*, *i.e.* if Z^+ designates a *set of terms sorts* then the term variable is called *Z*. The representation of the production rules by tables is nothing but an inverse BNF.

The next level below the general *sort Akton* comprises the *sorts Head*, *Body*, *Tail*, *CS*. Their properties have been informally defined in the previous chapter. The four

sorts establish a set *FA* of fundamental *Akton sorts*.

Definition A.1 (Fundamental Akton Sorts)

The set FA of fundamental akton sorts is defined as
FA := {Head, Body, Tail, CS}, where
abstract: FA→ {Akton}
abstract(x):= Akton, if x∈{Head, Body, Tail, CS}

An *Interface* at the second level of the specification hierarchy (see **Figure** 1) is either empty or not empty, represented by ε or ¬ε. The two elements establish a *set FI* of *fundamental interface sorts*.

Definition A.2 (Fundamental Interface Sorts)

The set FI of fundamental interface sorts is defined as
FI := {¬ε ,ε}, where
abstract: FI→ {Interface}
abstract(x):= Interface, if x∈{¬ε, ε}

Every *Akton term x* has two *Interfaces*, an *input* and an *output*, formally described by *in(x)* and *out(x)*. Depending on the level of concretization, the *Interfaces* may be refined by introducing more properties. This gives rise to a hierarchy of *Interface sorts*, similar to the hierarchy of *Akton sorts*. At the fundamental level, these functions have the following definition.

Definition A.3 (Fundamental Term Level Functions in, out)

The fundamental term level functions in, out: $FA^+ \rightarrow FI$ are inductively defined as:

$$in(z):=\begin{cases} \varepsilon, & if \quad z\in Head\cup CS \\ \neg\varepsilon, & if \quad z\in Body\cup Tail \\ in(x)/in(y), & if \quad z = (x/y) \\ in(x), & if \quad z = (x>y) \end{cases}$$

$$out(z):=\begin{cases} \varepsilon, & if \quad z\in Tail\cup CS \\ \neg\varepsilon, & if \quad z\in Head\cup Body \\ out(x)/out(y), & if \quad z = (x/y) \\ out(y), & if \quad z = (x>y) \end{cases}$$

The *in*- and *out-interfaces* of *Juxta*-related *Akton terms* are packed on top of each other, in the same way as the *Akton terms* themselves. This means that the *Juxta-symbol* is used for relating *Akton terms* as well as *Interface elements*. This way the order of the *Interface elements* matches the linear order of adjacent independent nodes, *i.e.*, the order serves as a local physical addressing scheme.

Definition A.4 (Fundamental Interface Terms)

The set of fundamental interface terms is defined as:
$FI \subseteq FI^*$
$\forall x,y \in FI^*$: *$(x/y) \in FI^*$, if*

x= \ y=	(x/y)	¬ε	ε
	¬ε	¬ε	¬ε
	ε	¬ε	ε

The third level of the hierarchy of *Akton sorts* introduces structural properties. Consequently, it is called the *structural level*. Concerning *Aktons* of *sort Body*, there are three *subsorts* representing the structures fork, join and link. They are called *Fork, Join and Link* accordingly.

As discussed in the previous chapter, the mapping of spatial structures to planar structures requires a cut, represented by the pair of *Akton sorts Down* and *Up*. Likewise, the complete sequencing of planar structures requires another pair of *Akton sorts Off* and *Set.* Thus, *Head* comprises the *subsorts Up* and *Set* and *Tail* the *subsorts Down* and *Off*.

Definition A.5 (Structural Akton Sorts)

The set A of structural Akton sorts is defined as
A := {Up, Set} ∪ {Down, Off} ∪ {Fork, Join, Link} ∪ {CS}, where
abstract: A→ {FA}
abstract(x):= Head, if x∈{ Up, Set},
abstract(x):= Tail, if x∈{ Down, Off},
abstract(x):= Body, if x∈{Fork, Join,Link},
abstract(x):= CS, if x=CS

At the third level of the specification hierarchy (see **Figure 1**) an *Interface* consists of a set of ordered elements of *sort Pin* or is empty. This *set* of structural interface sorts is designated by *I*.

Definition A.6 (Structural Interface Sorts)

The set I of structural interface sorts is defined as
I := {Pin, ε}, where
abstract: I→ {FI}
abstract(x):=¬ε, if x=Pin
abstract(x):=ε, if x=ε

Definition A.7 (Structural Term Level Functions in, out)

The term level functions in, out: $A^+ \rightarrow I^*$ are inductively defined as:

$$in(z):=\begin{cases} \varepsilon, & if \quad z\in Head\cup CS \\ Pin, & if \quad z\in Tail\cup Link \\ Pin, & if \quad z = Fork \\ Pin/Pin, & if \quad z = Join \\ in(x)/in(y), & if \quad z = (x/y) \\ in(x), & if \quad z = (x>y) \end{cases}$$

$$out(z):=\begin{cases} \varepsilon, & if \quad z\in Tail\cup CS \\ Pin, & if \quad z\in Head\cup Link \\ Pin/Pin, & if \quad z = Fork \\ Pin, & if \quad z = Join \\ out(x)/out(y), & if \quad z = (x/y) \\ out(y), & if \quad z = (x>y) \end{cases}$$

Definition A.8 (Structural Interface Terms)

The set of structural interface terms is defined as the smallest set satisfying:
$I \subset I^*$
$\forall x,y \in I^*$: *$(x/y) \in I^*$*

The *set* of *Akton terms* A^+ is a *subset* of FA^+. A first restriction of A^+ is that two dependent *terms* can only be *Next*-related if their adjacent *Interfaces* are identical. The other restriction is that every relation of *terms* must conform to a real topological structure.

A crossing for instance is characterized by the *cut terms D,* $D \in D^+$, and *U,* $U \in U^+$, with a separate *term B,* $B \in B^+$ in between. This gives rise to four special *term sorts*, designated by DB^+, BD^+, UB^+, BU^+ (see **Figure 2**). There are four of them because every crossing owns a chirality that can be either *left-* or *right-handed*. The planar structures of cycles and crosslinks on the other hand are characterized by the *cut terms O,* $O \in O^+$, and *S,* $S \in S^+$, which are not separated by another *term* (see **Figure 3**). A pair of *cut terms* can thus only be positioned either at the *left* or at the *right* side of a *B-term*, which results in four special *term sorts* OB^+, BO^+, SB^+, BS^+. Four more special *term sorts* OBO^+, OBS^+, SBO^+, SBS^+ need to be introduced because independent *cut terms* may occur at both sides of a *B-term*.

The *set* of production rules P consists of four *subsets* P_i, $i \in \{1,2,3,4\}$. P_1 lists the *fundamental rules*, P_2 the *planarizing rules*, P_3 *the linearizing rules* and P_4 the *twin-cut rules*.

Definition A.9 (Terms of the Akton Language)

The set of Akton terms A^+ *is inductively defined as the smallest set satisfying:*

$A^+ := B^+ \cup E^+ \cup T^+ \cup CS^+ \cup U^+ \cup D^+ \cup S^+ \cup O^+$

$Body \in B^+$, $Head \in H^+$, $Tail \in T^+$, $CS \in CS^+$, $Up \in U^+$, $Down \in D^+$, $Set \in S^+$, $Off \in O^+$

$\forall x,y \in A^+$: $(x>y) \in A^+$, *if* $out(x)=in(y)$ *and* $(x>y) \in P_i$

$\forall x,y \in A^+$: $(x/y) \in A^+$, *if* $(x/y) \in P_i$

$P_i := P_1 \cup P_2 \cup P_3 \cup P_4$

Definition A.10 (Fundamental Production Rules of the Akton Language)

The set of fundamental production rules P_1 *is defined as:*

x= \ y=	H	B	T	CS
H	H	B	B	H
B	B	B	B	B
T	B	B	T	T
CS	H	B	T	CS

x= \ y=	B	T
H	H	CS
B	B	T

Definition A.11 (Planarizing Production Rules of the Akton Language)

The set of planarizing production rules P_2 *is defined as:*

x= \ y=	B	U	D	BU	UB	DB	BD	CS
B	B	BU	BD	BU			BD	B
U	UB	U			UB			U
D	DB		D			DB		D
BU		BU						BU
UB	UB							UB
DB	DB							DB
BD			BD					BD
CS	B	U	D	BU	UB	DB	BD	CS

x= \ y=	D	B	BU	UB
U	CS	U		
B	D	B	BU	UB
DB		DB	B	
BD		BD		B

The complete definition of crossings (see **Figure 2**) requires the additional definition of the implicit *in-out*-relations between *U-* and *D-terms*. The following two definitions regard the mirrored structures alternatively crossing a *B-term* either from above or below.

Definition A.12 (Implicit Cut-Relations in Crossings)

The implicit cut-relations in crossings are defined as:

$\forall v,w,x,y \in A^+$:

$out(y) := in(v)$, *if* $(v/w>x/y) \in B^+$ *and* $y \in U^+$ *and* $v \in D^+$

or

$out(x) := in(w)$, *if* $(v/w>x/y) \in B^+$ *and* $x \in U^+$ *and* $w \in D^+$.

Similarly, the complete definition of cycles and crosslinks requires the additional definition of the implicit *in-out* relations between *O-* and *S-terms*. Recall that the feedback of a cycle is always located either above or below a *B-term* (see **Figure 3a**). A crosslink, on the other hand, is always located between two *B-terms* connecting an upper *B-term* with a lower *B-term* or vice versa (see **Figure 3b**).

Definition A.13 (Linearizing Production Rules of the Akton Language)

The set of linearizing production rules P_3 *is defined as:*

x= \ y=	B	S	O	BO	BS	OB	SB	CS
B	B	BS	BO	BO	BS			B
O	OB	B	O	OBO	OBS			O
S	SB	S	B	SBO	SBS			S
OB	OB	OBS	OBO	OBO	OBS			OB
SB	SB	SBS	SBO	SBO	SBS			SB
BS						B		BS
BO							B	BO
CS	B	S	O	BO	BS	OB	SB	CS

	(x>y)	O	B	BO	OB	BS	SB	OB	OBS	SBO	SBS
	S	CS	S								
	B	O	B	BO	OB	BS	SB	OBO	OBS	SBO	SBS
	BS		BS	B				OB		SB	
	SB		SB		B			BO	BS		
x=	BO		BO			B			OB		SB
	OB		OB				B			BO	BS
	SBS		SBS	SB	BS			B			
	SBO		SBO		BO	SB			B		
	OBS		OBS	OB			BS			B	
	OBO		OBO			OB	BO				B

y=

Definition A.14 (Implicit Cut-Relations in Cycles)
The implicit cut-relations in cycles are defined as:
$\forall v,w,x,y \in A^+$:
$out(v):=in(x)$, *if* $(v/w>x/y) \in B^+$ *and* $v \in S^+$ *and* $x \in O$ *or*
$out(w):=in(y)$, *if* $(v/w>x/y) \in B^+$ *and* $w \in S^+$ *and* $y \in O^+$.

Definition A.15 (Implicit Cut-Relations in Crosslinks)
The implicit cut-relations in crosslinks are defined as:
$\forall u,v,w,x,y,z \in A^+$: $out(v):=in(y)$, *if* $(u>x/y)/(v/w>z) \in B^+$ *and* $v \in S^+$ *and* $y \in O^+$ *or*
$out(v):=in(y)$,*if*$(u/v>x)/(w>y/z) \in B^+$*and* $v \in S^+$*and* $y \in O^+$.

The final structure to be defined by production rules is the twin-cut structures as outlined in section 2.4 of the previous chapter. As mentioned the twin-cut structures may be either *left-* or *right-twisted*. Changing the direction of the spatial as well as the planar cut gives rise to four different structures each. The left-twisted structures are depicted in **Figure 5**.

Definition A.16 (Twin-Cut Production Rules of the Akton Language)
The set of the twin-cut production rules P_4 is defined as:

y=

x=	(x>y)	BO	BS	DB	UB
	BU	BUBO	BUBS		
	BD	BDBO	BDBS		
	SB			SBDB	SBUB
	OB			OBDB	OBUB

y=

x=	(x/y)	SBDB	OBUB	SBUB	OBDB
	BUBO	B			
	BDBS		B		
	BDBO			B	
	BUBS				B

Definition A.17 (Implicit Twin-Cut Relations in Left-twisted Antiparallel Structures)
The implicit cut-relations in left-twisted antiparallel structures are defined as:
$\forall s,t,u,v,w,x,y,z \in A^+$: $out(t):=in(y)$ and $out(u):=in(x)$, *if* $(s/t>w/x)/(u/v>y/z) \in B^+$ and $t \in U^+$ and $y \in D^+$ and $x \in O^+$ and $u \in S^+$,
$\forall s,t,u,v,w,x,y,z \in A^+$: $out(x):=in(u)$ and $out(y):=in(t)$, *if* $(s/t>w/x)/(u/v>y/z) \in B^+$ and $y \in U^+$ *and* $t \in D^+$ and $u \in O^+$ and $x \in S^+$,
$\forall s,t,u,v,w,x,y,z \in A^+$: $out(y):=in(t)$ and $out(u):=in(x)$, *if* $(s/t>w/x)/(u/v>y/z) \in B^+$ and $y \in U^+$ and $t \in D^+$ and $x \in O^+$ and $u \in S^+$,
$\forall s,t,u,v,w,x,y,z \in A^+$: $out(t):=in(y)$ *and* $out(x):=in(u)$, *if* $(s/t>w/x)/(u/v>y/z) \in B^+$ and $t \in U^+$ and $y \in D^+$ *and* $u \in O^+$ and $x \in S^+$.

Definition A.18 (Implicit Twin-Cut Relations in Right-twisted Antiparallel Structures)
The implicit cut-relations in right-twisted antiparallel structures are defined as:
$\forall s,t,u,v,w,x,y,z \in A^+$: $out(t):=in(y)$ and $out(u):=in(x)$, *if* $(s/t>w/x)/(u/v>y/z) \in B^+$ and $u \in U^+$ and $x \in D^+$ and $y \in O^+$ and $t \in S^+$,
$\forall s,t,u,v,w,x,y,z \in A^+$: $out(x):=in(u)$ and $out(y):=in(t)$, *if* $(s/t>w/x)/(u/v>y/z) \in B^+$ and $x \in U^+$ and $u \in D^+$ and $t \in O^+$ and $y \in S^+$,
$\forall s,t,u,v,w,x,y,z \in A^+$: $out(y):=in(t)$ and $out(u):=in(x)$, *if* $(s/t>w/x)/(u/v>y/z) \in B^+$ and $x \in U^+$ and $u \in D^+$ and $y \in O^+$ and $t \in S^+$,
$\forall s,t,u,v,w,x,y,z \in A^+$: $out(t):=in(y)$ and $out(x):=in(u)$, *if* $(s/t>w/x)/(u/v>y/z) \in B^+$ and $u \in U^+$ and $x \in D^+$ and $t \in O^+$ and $y \in S^+$.

Single *Forks*, *Joins* and *Links* have a planar structure, and *multiple Links* are also planar. The structures of *multiple Forks* and *multiple Joins,* however, are always spatial, *i.e.* projecting them on a plane can only be achieved by means of *Down*/*Up-cuts*. A *multiple Fork* structure duplicates a structure of *multiple Links* into two identically ordered structures of *multiple Links*, and a *multiple Join* term does the reverse. In order to formally describe these structures, we need to introduce the separation functions *pre* and *suc*, which split a *Next*-relation into a preceding and a succeeding part.

Definition A.19 (Separation Functions pre, suc)
The functions pre, suc: $A^+ \rightarrow A^+$ *are defined as:*
$pre(x>y) := x$,
$suc(x>y) := y$

Definition A.20 (Multiple Link)
The set of link terms L^+ *is inductively defined as the smallest set satisfying:*
$L^+ \subset B^+$
$Link \in L^+$
$\forall x \in L^+$: $Link/x \in L^+$

Multiple Forks as well as *multiple Joins* can be real-

ized by either *left-hand* or *right-hand* twisting the otherwise identical structures. Accordingly, there are two *sets* of *multiple Forks* and of *multiple Joins*. They will be distinguished by the subscripts *l* and *r* indicating whether the twist is *left- or right-handed*.

Definition A.21 (Multiple Fork)

The sets of multiple fork terms are recursively defined as:

$F_l^+ \cup F_r^+ \subset B^+$,

$(Fork>Down/Link) > (Link/Up) \in F_l^+$,

$\forall x \in F_l^+$:$(pre(x)/(Fork>Down/Link) > (Link/suc(x)/Up))$

$(Fork > Link/Down) > (Up/Link) \in F_r^+$,

$\forall x \in F_r^+$:$(pre(x)/(Fork > Link/Down) > (Up/suc(x)/Link))$

Definition A.22 (Multiple Join)

The sets of multiple join terms are recursively defined as:

$J_l^+ \cup J_r^+ \subset B^+$,

$(Down/Link) > (Link/Up > Join) \in J_l^+$,

$\forall x \in J_l^+$:$(Down/pre(x)/Link > suc(x)/(Link/Up>Join))$

$(Link/Down) > (Up/Link > Join) \in J_r^+$,

$\forall x \in J_r^+$:$(Link/pre(x)/Down > suc(x)/(Up/Link>Join))$

Modularity, *i.e.* the capability of combining several modules into a single one, is an indispensable requirement for the design of complex systems. In *AA* modularity is easily incorporated because all modules are *Aktons*, and every *Akton term*, how big it ever may be, can be concealed into a single *Akton*. Concealing means hiding the structure of an *Akton term* into an *Akton* while preserving the visibility of the *input* and the *output*. The new *Akton* is added to set *A,* and of course needs to be provided with a distinct name. In contrast, regarding conventional digital programming languages, modularization and information hiding can be quite a problem [5].

Definition A.23 (Function *conceal*)

The function conceal: $A^+ \rightarrow A$ *is defined as:*

$conceal(x) := y$, *if not* $x \in A$

In order to densify the *Akton* expressions later on we introduce a *count* function for sequences of identical *Next-terms* and a *count* function for regular *Juxta-terms*. *Next-counts* are represented by symbol '*'(called *star*) and Juxta-*counts* by symbol '^'(called *roof*). Syntactically, both bind stronger than *Next and Juxta.*

Definition A.24 (Counting Functions *, ^ of Next- and Juxta-Terms)

The counting function $*: \mathbf{N} \times A^+ \rightarrow A^+$ *is inductively defined as:*

$(n+1)*x = n*x>x$, $1*x = x$, $n \in \mathbf{N}$, $x \in A^+$

The counting function $\hat{}: A^+ \times \mathbf{N} \rightarrow A^+$ *is inductively defined as:* $x\hat{}(n+1) = x\hat{}n/x$, $x\hat{}1 = x$, $n \in \mathbf{N}$, $x \in A^+$

We also introduce the *counting function* ^ for sequences of regular *Interface terms*.

Definition A.25 (Counting Function ^ of Interface-Terms)

The counting function $\hat{}: I^* \times \mathbf{N} \rightarrow I^*$ *is inductively defined as:*

$x\hat{}(n+1) = x\hat{}n/x$, $x\hat{}1 = x$, $n \in \mathbf{N}$, $x \in I^*$

4. Dependency Preserving Term Replacements

The structure of a given nodal network can be modified in different ways without affecting the dependencies between the *terms*. Formally the modifications are achieved *by term replacement* according to the rules of Tab. 1. The rules say that the *left term* may be replaced by the *right term* provided that the constraint at the *right* side holds. The "↔"-symbol says that the terms are mutually replaceable.

Table 1: *Dependency preserving term replacement rules*

a. Link-Rules:	$x \leftrightarrow (y>x)$, *if* $in(y)=in(x)$
	$x \leftrightarrow (x>y)$, *if* $out(y)=out(x)$
b. Expansion-Rules:	$x \leftrightarrow (y/x)$, *if* $y \in CS^+$
	$x \leftrightarrow (x/y)$, *if* $y \in CS^+$
c. Associativity-Rules:	$((x>y)>z) \leftrightarrow (x>(y>z))$, *if true*
	$((x/y)/z) \leftrightarrow (x/(y/z))$, *if true*
d. Distributivity:	$((w>x)/(y>z)) \rightarrow (w/y>x/z)$, *if true*
	$(w/y>x/z) \rightarrow ((w>x)/(y>z))$, *if* $out(w)=in(x)$
e. Connectivity-Rules:	$((w>x)/(y>z)) \leftrightarrow (w>x/y>z)$, *if* $out(x)=\varepsilon$ *and* $in(y)=\varepsilon$
	$((w>x)/(y>z)) \leftrightarrow (y>w/z>x)$, *if* $in(w)=\varepsilon$ *and* $out(z)=\varepsilon$
	$w,x,y,z \in A^+$

The link-rules a. add a *Link term y* to a *term x* or delete it. The *term y* may either precede or succeed *term x* as stated by the two rules. The constraints are that the output of the left *term* must fit the input of the right *term*. Usually term *y* will just consist of a strip of *Links*. The expansion-rules b. place or remove a *dead term y*, *i.e.* a neutral place, above or below a *term x*. *Term x* is of *sort* A^+, *term y* is of *sort* CS^+. Both expansion-rules play an important role in the layout process. The associativity-rules c. modify the structure of *Next-* and *Juxta*-related *terms*. The first of the distributivity-rules d. states that distributivity of *Juxta* over *Next* always holds while the other rule states that distributivity of *Next* over *Juxta* is restricted. The connectivity-rules e. splice two independent *Juxta-terms* into a single *term* or vice versa.

5. Abstract Structural Models

The properties of abstract *AA* are exemplified by four symbolic nodal structures. Each of the descriptions is accompanied by an *AA*-program by which every struc-

ture can completely be reconstructed.

5.1. Tetrahedron

The first example (see Figure 6) deals with a tetrahedron showing in detail the successive steps of mapping from the spatial structure to the linear program. In a first step a rear edge of the tetrahedron is cut, as marked by a thin red line, and the free ends are marked by an *Up/Down*-pair.

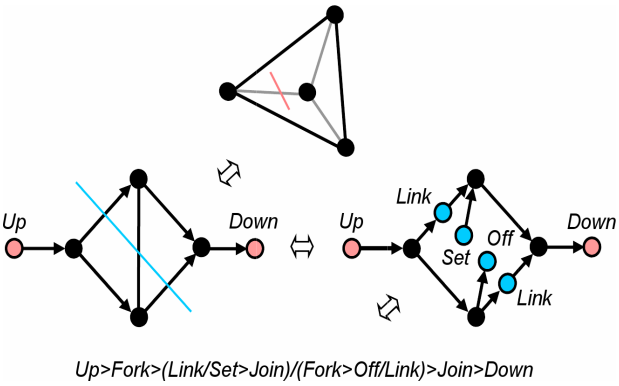


Figure 6. The mapping of a tetrahedron from the spatial structure to a linear *AA*-program needs three steps. It is first planarized by an *Up/Down* cut along the red line, then fully oriented by a *Set/Off* cut along the blue line, and finally turned into an *AA- program*.

This makes it possible to spread the tetrahedron on a plane, and to orient the planar structure from *left* to *right* according to the direction introduced by the *Up/Down*-plane, and to orient the planar structure from *left* to *right* according to the direction introduced by the *Up/Down*-pair. The planar structure is then cut again, as marked by a long thin blue line. While the cuts of both outer edges are healed by inserting *Links* the cut of the crosslink is marked by a *Set/Off*-pair, all shown in blue. This provides the crosslink with a unique direction (see **Figure 6**). (A reversely ordered *Off/Set*-pair would of course reverse the direction of the crosslink.) The resulting structure is represented by a linear program.

5.2. Helix and Sheet

The next two examples have been selected in order to indicate the relation between *AA* and the as yet unknown protein programming language, which programs the spatial structure of proteins by chains of amino acids [1]. Since the vocabulary of *AA* is just derived from a few general principles it can be conjectured that the amino acids are also describable by *AA-expressions*. The structure **(a)** of **Figure 7** represents a model of two loops of a *right*-handed α-helix. Since a α-helix is a spatial structure, it takes *Up/Down*-pairs to planarize it and *Set/Off*-pairs to fully orient it from *left* to *right*. The structure (b) of **Figure 7** represents a model of a ß-pleated sheet. Since this structure is planar, it only takes *Set/Off*-pairs in order to stretch it into a programming code.

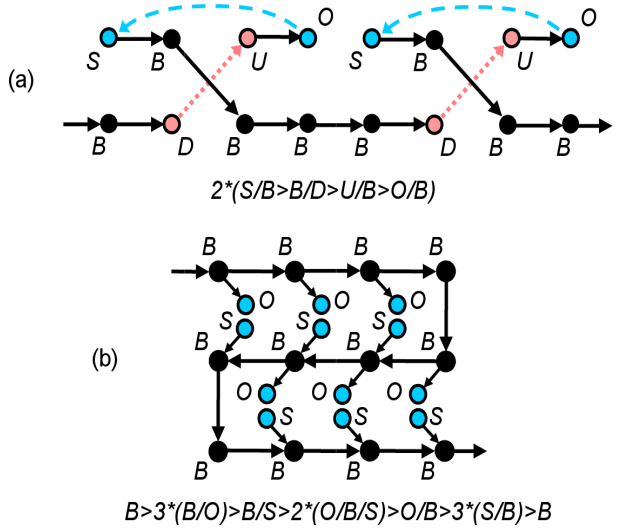


Figure 7. Model and *AA-program* of a right-handed α-helix (a), and model and *AA*-program of a β-sheet (b.)

5.3. Modeling DNA

A first particular application of twin-cuts concerns the modeling of the genetic code.

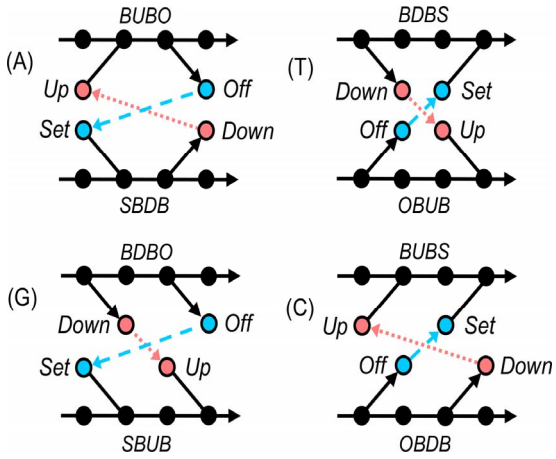


Figure 8. Nucleotide models of the four elements of DNA-code. Akton *term* (A) is pair-wise complementary to *Akton term (T)*, and *Akton term (G)* is pair-wise complementary to *Akton term (C)*.

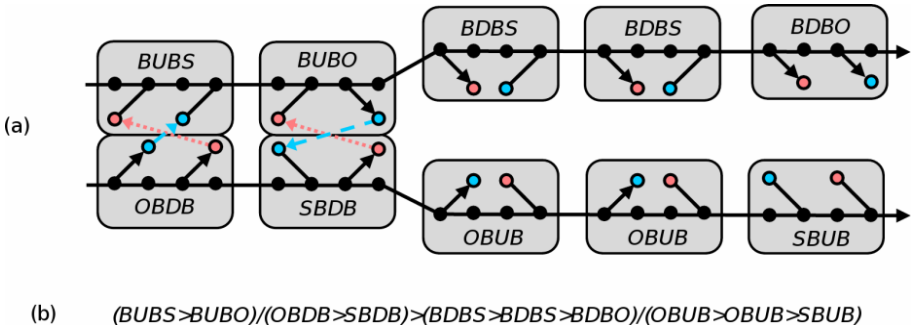


Figure 9. Model of a double stranded DNA, partially split into two single strands modeling RNA. Figure (a) shows the planar *AA*-model, Figure (b) the linear *AA-expression*.

As well-known, the basic information on the construction of all organisms are expressed by a 4-symbol programming language, where the symbols represent the nucleobases called Adenine (A), Guanine (G), Cytosine (C) and Thymine (T) [6]. Here we will use the same abbreviations for the nucleotides, whereupon each nucleobase is extended by a piece of backbone. An important feature of the nucleotides is that they are pair-wise complementary, *i.e.* A matches with T, and G matches with C. The twin-cuts of *AA* offer exactly the same property and thus are perfectly suited to model DNA. Since *AA* distinguishes between above and below, there are two representations for each of the four nucleotides. This is visualized by **Figure 8**, where the four twin-cuts of **Figure 5** are now described by *akton terms* according to the production rules P_4.

Assuming that Adenine (A*)* is represented by the *Akton term BUBO* then Thymine (T*)* is represented by *SBDB*. Likewise, assuming that Guanine (G*)* is represented by *BDBO* then Cytosine (C*)* is represented by *SBUB*. **Figure 8** models the four elements of DNA-code. The elements are pairwise complementary to each other. Exchanging *U* and *D* (the red balls) as well as *S* and *O (the blue balls)* turns *term* (A) into *term* (T), and *term* (G) into *term* (C). This is exactly the matching property of Adenine/Thymine-pair and the Guanine/Cytosine-pair.

The four elements of the DNA-code are perfectly suited to describe a double stranded helix as well as RNA. **Figure 9a** shows an example and **Figure 9b** the *AA*-program.

A second particular feature of the twin-cuts is that their *left-* or *right-twist* is suited to model the chirality of a double helix. This statement is substantiated by the following arguments: Since topological nodes have a finite albeit unknown volume and twin-cuts are crossings of two pairs of nodes, twin-cuts do have a natural skew. As shown in **Figure 10**, a *left-twisted* twin-cut causes a *right-twisted* skew between the strands and vice versa. It is this skew that causes two strands which are interconnected by twin-cuts to turn into a double helix.

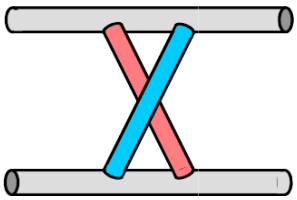


Figure 10. Simplified sketch of a twin-cut visualizing its skew.

Another example of a twin-cut structure is the circuit diagram of an *SR*-Flipflop (see **Figure 12**) which will be treated in section 6.2.

6. Concretizing *AA*: Symbolic Systems

The language of *AA*, as defined up to this point, describes the topological structure of *abstract discrete spatial systems*. It can now step by step be concretized towards special discrete systems by introducing new *Aktons* as *subsorts* of the *abstract Akton sorts*. The new *subsorts* inherit the properties of their hierarchical ancestors and may be provided with additional properties. However, none of the additional properties may ever conflict with the inherited properties. There are three main ways how to concretize *abstract AA*. Most trivially, it could be done by introducing new *subsorts* and designating them only symbolically, *i.e.* without adding new properties. Typical examples are wiring diagrams of analog or digital circuitry.

The new *akton sorts* can be provided with functionality by extending the *Interface* hierarchie giving rise to functional systems. Finally, the *Akton* and the *Interface* hierarchie can be provided with a metric giving rise to concrete spatial systems. Each of the three modes will be studied in the sequel.

6.1. Digital Circuit Description

As well-known, every *binary* function can be realized by a single *sort Nand* or by a single *sort Nor*. Usually however, several *sorts* are applied. Here, we introduce the *subsorts And*, *Or*, *Not* and *Wire*, where *Not* denotes the *inversion* function and *Wire* the *1*-function. *And* and *Or* are *subsorts* of *Join*, and *Not* and *Wire* are *subsorts* of *Link*. The extension of the *Akton* hierarchy by concrete *subsorts* is depicted in **Figure 11(a)**. The set of *digital subsorts* is designated by *dA*.

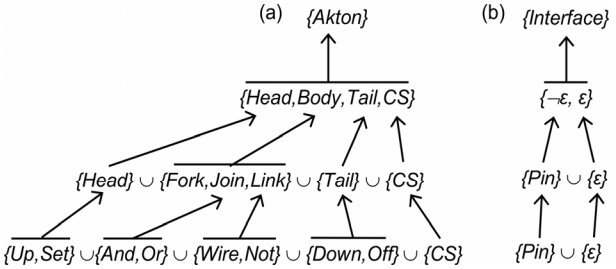


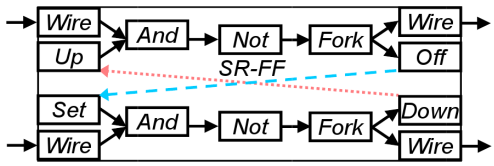
Figure 11. Extending *sort Join* of the akton hierarchy (a) by the subsorts *And* and *Or*, and sort *Link* by the subsorts *Wire* and *Not*, keeping the *Interface* hierarchy (b) unchanged, turns *AA* into *a digital circuit description* language.

Definition D.1 (Digital Akton Sorts)
The set dA of digital akton sorts is defined as:
$dA:=\{Up, Set\} \cup \{Fork\} \cup \{And, Or\} \cup \{Wire, Not\} \cup \{Down, Off\} \cup \{CS\}$, *where*
$abstract: dA \rightarrow A$,
$abstract(x):= Join, if\ x \in \{And, Or\}$
$abstract(x):= Link, if\ x \in \{Wire, Not\}$
$abstract(x):= x, if\ x \in \{Up, Set, Fork, Down, Off, CS\}$
Definition D.2 (Digital Akton Terms)
The set of digital akton terms dA^+ *is inductively defined as the smallest set satisfying:*
$dA \subset dA^+$
$\forall x,y \in dA^+: (x>y) \in dA^+$
$\forall x,y \in dA^+: (x/y) \in dA^+$

The properties of the *digital circuit description language* defined on the *set of digital akton terms* dA^+ are subsequently demonstrated by *three digital circuits* and their *description* by an *AA*-program.

6.2. SR-Flipflop

The *SR*-Flipflop shown in **Figure 12** serves to emphasize the important property of *AA* to analytically describe feedback circuits. With this property *AA* overcomes the severe restriction of register-transfer-level programming languages which only describe the logical expressions between two storage cycles [7].



SR-FF := (Wire/Up>And>Not>Fork>Wire/Off)/
(Set/Wire>And>Not>Fork>Down/Wire)

Figure 12. Symbolic Circuit diagram and AA-program of an SR-Flipflop.

6.3. Half- and Full-Adder

The examples of a half- and a full-adder shown in **Figure 13** serve to demonstrate how more complex systems can be built up from low level systems either by designating *Akton terms* by abbreviations or by concealing them into *Aktons*. Systems of arbitrary complexity can thus be treated just as every simple system.

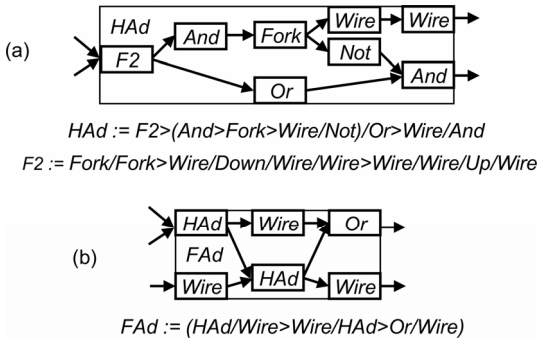


Figure 13. Symbolic Circuit diagrams and *AA-programs* of a half-adder *HAd* (a), and a full-adder *FAd* (b). The structure and the *Akton term* on top of Figure 13 show the circuit diagram of the half-adder *HAd* (a), and those at the bottom the circuit diagram of the full adder *FAd* (b). Both *Akton terms* are of *sort* B^+. *F2* designates a *two-Fork structure* doubling 2 inputs into 4 outputs.

The structure and the *Akton term* on top of **Figure 13a** show the circuit diagram and the *AA*-program of a half-adder *HAd*. Figure 13b at the bottom depicts the circuit diagram and *AA*-program of a full adder *FAd*. Both *Akton terms* are of sort B^+. *F2* designates a two-*Fork* structure doubling two inputs into four outputs.

7. Concretizing *AA*: Functional Systems

In the previous section we concretized *AA* symbolically, *i.e.* only by extending the *Akton* hierarchy by additional *Akton sorts*, however without providing them with any extra properties. This step alone was sufficient to create a

programming language for symbolic system description and design.

We now proceed to provide the newly introduced *Akton sorts* with functional properties. This enhances *AA* to a *general data processing* language. The enhancement is achieved by introducing data values as subsorts of the interface sort *Pin* and by defining functions between the input and the output of the *Aktons*. This way several kinds of functionality can be implemented, e.g. digital or analog functions or even both together. Moreover, because *AA* can be equipped with all the elementary functions of analog or digital circuitry and because these functions can arbitrarily be composed to more complex functions, a plethora of low or high level programming languages can be created this way.

It should also be noted that *abstract AA* does not impose any restrictions on the system behaviour. Since *abstract AA* does not make use of the notion of states, the execution of an *AA*-program behaves flow-controlled or, if *digital data processing* is introduced, as data driven [8]. However, the data driven behaviour can easily be turned into a state driven one by starting the evaluation of a succeeding akton only after the output of the preceding aktons is fully defined. A clock driven behaviour, *i.e.* the behaviour of most computers, can then be achieved by supplying each *akton term* with a storage function and by fitting the *Akton* evaluation time into the clock cycle.

7.1. Digital Data Processing

In this section, we concentrate on data driven *digital data processing*. To this end, we extend *sort Join* of the *Akton*-hierarchy not just by the subsorts *And* and *Or*, and sort *Link* by the subsorts *Wire* and *Not* (see figure 14a).

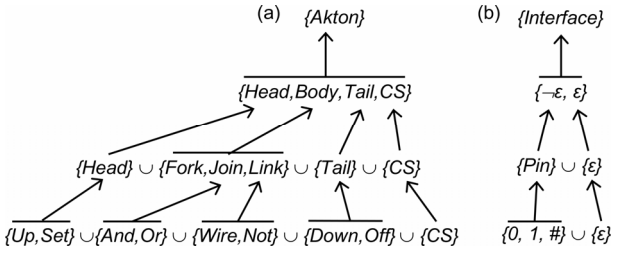


Figure 14: Extending sort *Join* by the subsorts *And* and *Or*, and sort *Link* by the subsorts *Wire* and *Not*, turns *AA* into a digital circuit description language (a). Further extending sort *Pin* of the *Interface* hierarchie (b) by the subsorts 0,1,# and defining the functions *in* and *out* by them, as done by definition D.5, generates a digital data processing language.

In addition, we now expand sort *Pin* of the interface-hierarchy (**Figure 14b**) by the digital values (*0, 1, #*), where value # indicates the state of the *Pin* before processing. Each of the functions contained in an *Akton* can only be evaluated, if its individual input data are available. Moreover, since the evaluation of the functions takes a different finite amount of time, the output of an *Akton* is always delayed and if there are several *output Pins,* they are never exactly synchronized.

Definition D.3 (Digital Interface Sorts)
The set of digital interface sorts dI is defined as:
$dI := \{0, 1, \#\} \cup \{\varepsilon\}$, *where*
$abstract: dI \rightarrow I$
$abstract(x) := Pin$, *if* $x \in \{0, 1, \#\}$
$abstract(x) := \varepsilon$, *if* $x = \varepsilon$

Definition D.4 (Digital Interface Terms)
The set of digital interface terms dI^* *is inductively defined as the smallest set satisfying:*
$dI \subset dI^*$,
$\forall x,y \in dI^*: (x/y) \in dI^*$

Definition D.5 (Digital Functions *in, out*)
The digital functions in, out: $dA^+ \rightarrow dI^*$ *are defined as:*

$out(And) := 1$,	if $in(And)=1/1$
$out(And) := 0$,	if $(in(And)=0/x$ or $in(And)=x/0)$
$out(And) := \#$,	if $(in(And)=\#/x$ or $in(And)=x/\#)$
$out(Or) := 0$,	if $in(Or)=0/0$
$out(Or) := 1$,	if $(in(Or)=1/x$ or $in(Or)=x/1)$
$out(Or) := \#$,	if $(in(Or)=\#/x$ or $in(Or)=x/\#)$
$out(Not) := 1$,	if $in(Not)=0$
$out(Not) := 0$,	if $in(Not)=1$
$out(Not) := \#$,	if $in(Not)=\#$
$out(Wire) := x$,	if $in(Wire)=x$
$out(Fork) := x/x$,	if $in(Fork)=x$
$out(Head) := x$,	if $in(Head)=\varepsilon$
$out(Tail) := \varepsilon$,	if $in(Tail)=x$
	$x \in \{1,0,\#\}$

Establishing the transmission of data between *Next*-related *Akton terms* extends *AA* from a description language of static systems to a description language of dynamic systems. In particular, this renders it possible to describe positive or negative feedback, *i.e.* either the storage of data or the repetition of functions. The following definition can therefore be regarded as a basic rule of computation.

Definition D.6 (Digital Data Transmission)
The digital data transmission is defined as:
$\forall x,y \in dA^+: in(y) := out(x)$, *if* $(x>y)$

7.2. Systems Behaviour

The behaviour of linear *AA*-programs like those of the half-adder or the full-adder is immediately understandable just by providing them with data and then tracing their execution step-by-step. However, a cyclic program is not that simple. For this reason we inspect the behaviour of a feedback cycle as depicted in **Figure 15**. Part (a) shows the structure of the feedback cycle and its program, part (b) the behaviour. Initially, every *Akton* is in the undefined state *#*. Supplying state *0* to the *input*, *i.e.*

out(Set1):=0, results in a steady state, where all *Aktons* are defined. Recall that according to Def. A.14 *Off* transfers its *input* to the *output* of *Set*. If subsequently state *1* is supplied to the *input*, *i.e. out(Set1):=1*, the feedback cycle starts oscillating.

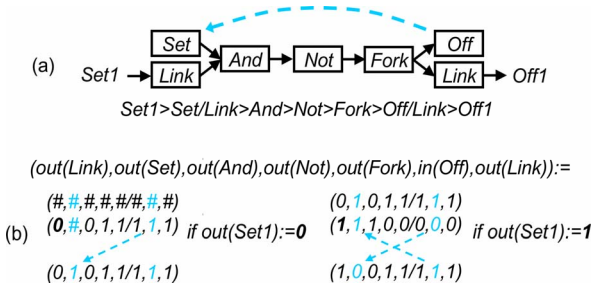


Figure 15: Structural and behavioural representation of a feedback circuit. Part (a) depicts the planar and linear representation of the circuit. Part (b) describes the states of the circuit components. In order to enter into a continuous loop the circuit first needs to be initialized by *out(Set1)*:=0 followed by *out(Set1):=1*.

8. Concretizing *AA*: Metric Systems

Recall that a correct *AA*-expression completely describes the topological structure of a physical system no matter, whether the structure is spatial, planar or linear, and no matter, whether the system representation is abstract or concrete. In order to fully reconstruct the metric of a given physical system from its abstract topological structure, we only have to reintroduce the original components and to eliminate the spatial cuts. This is the novel benefit of *AA* which has not been available up to now. However, our actual objective is more ambitious: We are aiming at a powerful tool to construct new systems and to adapt their concrete structures to arbitrary technical requirements. This makes it necessary to introduce a frame of reference by which the metric of the components, *i.e.* their shape and size, can consistently be defined.

A simple frame of reference can be introduced by assuming all *basic Aktons* having the same quadratic respectively cubical size.

The dimensionality of the frame of reference can be expressed by the directions a wayfarer would have to take to follow the directions of the *AA*-structure. Three directions are needed for a planar frame, *i.e. straight*, *left* and *right*, and two more for a spatial frame, *i.e. up* and *down*. Since there is no difference between a planar and a spatial frame of reference except for the number of directions, we confine ourselves to a planar one.

8.1. Rectangular Metric Systems

The planar directions are introduced into *AA* by means of structured subsorts of the *sorts Fork*, *Join* and *Link*.

There are three basic aktons to each subsort, *i.e.* $\{F_{lr},F_{ls},F_{sr}\}$, $\{J_{lr},J_{ls},J_{sr}\}$, $\{L_s,L_l,L_r\}$, where *F, J, L* are abbreviations of *Fork*, *Join* and *Link*, and the subscripts *s*, *l*, *r* are abbreviations of *straight*, *left* and *right*. (In a 3-dimensional reference frame the number of *basic Akton* would rise to 10 for subsorts of *Fork* and *Join* and to 5 for subsorts of *Link*.) The extended hierarchy of *Akton sorts* is shown in **Figure 16 (a)**. A uniform metric is achieved by extending the interface of sort ε by a subsort *Gap* representing an empty place with the same width as sort *Pin,* as shown in **Figure 16 (b)**.

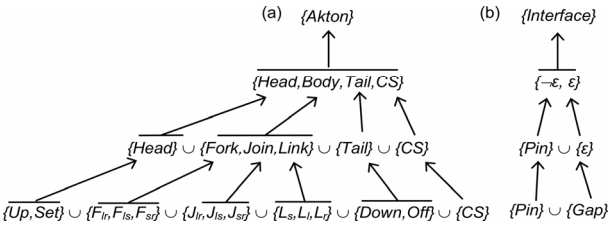


Figure 16. Extending the sorts *Fork*, *Join* and *Link* by structured metric subsorts (a) and introducing *Gap* as an identically sized companion of *Pin* (b) turns *AA* into a metric spatial programming language.

An arbitrary metric *Akton term* built up from quadratic basic components will generally not have a quadratic shape. However, an *Akton term* may always be given a rectangular shape by adding or deleting *basic Akton terms* being subsorts of *CS*. Without loss of generality it can therefore be assumed that every metric *Akton term* can always be given a rectangular shape.

Definition M.1 (Metric Akton Sorts)
The set of metric akton sorts mA is defined as:
$mA := \{Head\} \cup \{F_{lr}, F_{ls}, F_{sr}\} \cup \{J_{lr}, J_{ls}, J_{sr}\} \cup \{L_s, L_l, L_r\} \cup \{Tail\}$, *where*
$abstract: mA \rightarrow A$,
$abstract(x) := Fork$, *if* $x \in \{F_{lr}, F_{ls}, F_{sr}\}$,
$abstract(x) := Join$, *if* $x \in \{J_{lr}, J_{ls}, J_{sr}\}$,
$abstract(x) := Link$, *if* $x \in \{L_s, L_l, L_r\}$

As defined in A.7, an abstract structural interface is either empty or contains *Pins*. With the metric refinement an interface of an *Akton term* may now contain *Pins, Gaps* or both.

Definition M.2 (Metric Interface Sorts)
The set of metric interface sorts mI is defined as:
$mI := \{Pin\} \cup \{ Gap \}$, *where*
$abstract: mI \rightarrow I$,
$abstract(x) := x$, *if* $x = Pin$,
$abstract(x) := \varepsilon$, *if* $x = Gap$

Definition M.3 (Metric Interface Terms)
The set of metric interface terms mI^+ *is inductively defined as the smallest set satisfying:*
$mI \subset mI^+$

$\forall x,y \in mI^+: (x/y) \in mI^+$

On the other hand, by introducing a metric each *Akton term* now attains an individual size. Even the adjacent sides of two *Next*-related *Akton terms* $x>y$ may differ in size, because of different numbers of *Gaps* above and below their proper interfaces. These *Gaps* can formally be removed by introducing two functions, called *atrim* and *btrim*, where *atrim* eliminates all *Gaps* above a first *Pin* and *btrim* all *Gaps* below a last *Pin*. The function composition of *atrim* and *btrim*, *i.e.* the function *trim,* produces an interface beginning and ending with a *Pin*. This *Interface* can be interpreted as a plug. Plugging the *Interfaces* of two *Next-related terms* $x>y$ then means to shift both *terms* into their correct relative *Juxta*-position.

Definition M.4 (Functions *atrim, btrim, trim*)

The functions atrim, btrim, trim: $mI^+ \rightarrow mI^+$ *are inductively defined as:*

$atrim(i) := atrim(j)$ *while* $i=Gap/j$,

$btrim(i) := btrim(j)$ *while* $i=j/Gap$

$trim := atrim \circ btrim$.

In order to extract the metric interfaces of the four sides of a rectangular *Akton term* we introduce the functions s_i. The sides are clockwise enumerated, s_0 denoting the left-hand side. We first define the sides of the *basic Aktons*.

Definition M.5 (Functions s_i)

The functions $s_i: mA^+ \rightarrow \{side_i\}$, $side_i \in mI^+$, $i \in \{0,1,2,3\}$ *are defined as:*

$$s_i(z) := \begin{cases} \{Gap,Gap,Pin,Gap\}, & \text{if } z=Head \\ \{Pin,Gap,Gap,Gap\}, & \text{if } z=Tail \\ \{Gap,Gap,Gap,Gap\}, & \text{if } z=CS \\ \{Pin,Pin,Gap,Pin\}, & \text{if } z=F_{lr} \\ \{Pin,Pin,Pin,Gap\}, & \text{if } z=F_{ls} \\ \{Pin,Gap,Pin,Pin\}, & \text{if } z=F_{sr} \\ \{Gap,Pin,Pin,Pin\}, & \text{if } z=J_{lr} \\ \{Pin,Pin,Pin,Gap\}, & \text{if } z=J_{ls} \\ \{Pin,Gap,Pin,Pin\}, & \text{if } z=J_{sr} \\ \{Pin,Gap,Pin,Gap\}, & \text{if } z=L_s \\ \{Pin,Pin,Gap,Gap\}, & \text{if } z=L_l \\ \{Pin,Gap,Gap,Pin\}, & \text{if } z=L_r \end{cases}$$

Next we define the structural relations between the *output* and the *input* of each *metric subsort*.

Definition M.6 (Metric Functions *in, out*)

The metric functions in, out: $mA^+ \rightarrow mI^+$ *are defined as:*

$out(L_s) := s_2$, *if* $in(L_s) = s_0$

$out(L_l) := s_1$, *if* $in(L_l) = s_0$

$out(L_r) := s_3$, *if* $in(L_r) = s_0$

$out(F_{ls}) := s_1/s_2$, *if* $in(F_{ls}) = s_0$

$out(F_{sr}) := s_2/s_3$, *if* $in(F_{sr}) = s_0$

$out(F_{lr}) := s_1/s_3$, *if* $in(F_{lr}) = s_0$

$out(J_{sr}) := s_2$, *if* $in(J_{ls}) = s_1/s_0$

$out(J_{ls}) := s_2$, *if* $in(J_{sr}) = s_0/s_3$

$out(J_{lr}) := s_2$, *if* $in(J_{lr}) = s_1/s_3$

$out(Head) := s_2$

$in(Tail) := s_0$

Definition M.7 (Rectangular Akton Terms)

The set of rectangular akton terms mA^+ *is inductively defined as the smallest set satisfying:*

$mA \subset mA^+$

$\forall x,y \in mA^+: (x>y) \in mA^+$,

if $trim(out(x)) = trim(in(y))$

In order to fold a network of rigid rectangular *Aktons* into a planar area, we need a means to turn the *Akton terms* into another direction. This can be achieved by introducing two tilt functions *tl* and *tr,* where *tl* turns an *Akton term* orthogonally to the *left* and *tr* orthogonally to the *right*.

Definition M.8 (Tilt Functions tl (anticlockwise), tr (clockwise))

The tilt functions tl, tr: $mA^+ \rightarrow mA^+$ *are defined as:*

$t(x>y) = t(x)>t(y)$,

$t(x/y) = t(x)/t(y)$, $t \in \{tl,tr\}$,

$tl(tr(x)) = tr(tl(x)) = x$

$tl(tl(x)) = tr(tr(x))$

Assigning input/output directions to the *Body Aktons* extends them to three *subsorts* each. The *basic sorts* of structured *metric Aktons* are depicted in **Figure 17**.

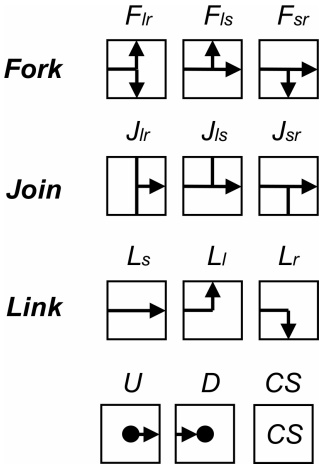


Figure 17. Basic sorts of structured metric Aktons. There are three subsorts of *Fork*, *Join* and *Link* each, which differ in the direction they proceed. Their paths turn either to the left, to the right or proceed straight. In addition, the basic metric aktons of *D (Down)*, *U (Up)* and *CS* are shown. The metric aktons *Set* and *Off* are skipped.

Abstract multiple Links, *multiple Forks* and *multiple Joins* have already been defined by Defs. A.21, A.22 and A.23. These structures are important in the layout process of digital systems.

According to the rectangular metric being assumed here, there are three structures of *metric multiple Links*, which can be generated making use of the *basic metric subsorts* L_s, L_l, L_r. While a *straight multiple Link* can just be generated by *Juxta*-relating *Links* to columns and then *Next*-relating the columns to strips of any finite length, tilted multiple structures need to be realized by *Juxta*-related structures of single chains of ascending and descending length which together form a square. The centerpiece of a *left-tilted* chain is an element of sort L_l (as shown later on in **Figure 19(a)**) and the centerpiece of *a right-tilted* chain is an element of sort L_r.

Definition M.9 (Metric Multiple Links)

The sets of metric multiple Links are recursively defined as:

$mL^+ \subset mA^+$

$mL_l^+ \cup mL_r^+ \subset mL^+$

$x(i) \in mL_l^+$

$x(1) := L_l$

$\forall i \in \mathbf{N}$: $x(i) := x(i)/((i-1)*L_s > L_l > tl((i-1)*L_s))$, *if* $1 \geq i$

$x(i) \in mL_r^+$

$x(1) := L_r$

$\forall i \in \mathbf{N}$: $x(i) := x(i)/((i-1)*L_s > L_r > tr((i-1)*L_s))$, *if* $1 \geq i$

Metric *multiple Fork* as well as *Join* structures have some special features. Firstly, a *metric multiple Fork* cannot be constructed from a basic element F_{lr}, because the two outputs of such a structure are ordered reversely. The same applies for the two inputs of a *metric multiple Join* if it constructed by a basic element $J_{lr.}$. Secondly, as already stated by Defs. A.21 and A.22, regular *abstract multiple Forks* and *Joins* are built up by crossings and therefore are either *left-* or *right-handed*. This chirality is preserved, if a metric is introduced. However, coming along with the metrisation is a path orientation which in case the chirality is *left-handed* is either oriented *straight* or *left-tilted*, and in case it is *right-handed* is either oriented *straight* or *right-tilted*. This gives rise to two *multiple Fork* constructs and two *multiple Join* constructs as represented in **Figure 18**.

Figure 18 further demonstrates that the constructs can be concealed to individual *Aktons* (see A.23), and since each side of the *Aktons* contains at most one *Pin* and a *via* they can be reduced to unit square size. This way the concealed structures get the appearance of a combination of a *Link* and a *via*. The concealed and minimized structures are designated by $F_{l,l}$ and $F_{r,r}$, resp. $J_{l,l}$ and $J_{r,r}$, where the first index defines the chirality of the *via* and the second the orientation of the *Link*.

$F_{l,l} := conceal(tl(D/L_s)/(F_{ls} > L_l))$, $F_{l,l} \in DB^+$

$s_i(F_{l,l}) := \{Pin, Pin, Gap, Gap\}$

$F_{r,r.} := conceal((F_{sr} > L_r)/tr(L_s/D))$, $F_{r,r} \in BD^+$

$s_i(F_{r,r}) := \{Pin, Gap, Gap, Pin\}$

$J_{l,l} := conceal((tr(U/L_s)/(tr(L_l) > J_{ls}))$, $J_{l,l} \in UB^+$

$s_i(J_{l,l}) := \{Gap, Pin, Gap, Pin\}$

$J_{r,r} := conceal((tl(L_r) > J_{sr})/tl(L_s/U))$, $J_{r,r} \in BU^+$

$s_i(J_{r,r}) := \{Gap, Gap, Pin, Pin\}$

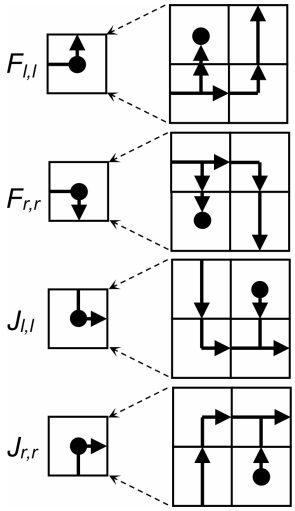


Figure 18. A concealed structure can be scaled down into a unit sqare rectangle if no *Pin* and no *via* points in the same direction.

Definition M.10 (Metric Multiple Forks)

The sets of metric multiple Forks are recursively defined as:

$mF^+ \subset mA^+$

$mF_{l,l}^+ \cup mF_{r,r}^+ \subset mF^+$

$x(i) \in mF_{l,l}^+$

$x(1) := F_{l,l}$

$\forall i \in \mathbf{N}$:

$x(i) := x(i)/((i-1)*L_s > F_{l,l} > tl((i-1)*L_s)) > tl(L_s\text{^}i/CS)/U\text{^}i$, *if* $1 \geq i$

$x(i) \in mF_{r,r}^+$

$x(1) := F_{r,r}$

$\forall i \in \mathbf{N}$:

$x(i) := x(i)/((i-1)*L_s > F_{r,r}^+ > tr((i-1)*L_s)) > tr(CS/L_s\text{^}i)/U\text{^}i$, *if* $1 \geq i$

Definitions M.11 (Metric Multiple Joins)

The sets of metric multiple Joins are recursively defined as:

$mJ^+ \subset mA^+$

$mJ_{l,l}^+ \cup mJ_{l,l}^+ \subset mJ^+$

$x(i) \in mJ_{l,l}^+$

$x(1) := J_{l,l}$

$\forall i \in \mathbf{N}$:

$x(i) := x(i)/((i-1)*L_s > tl(J_{l,l}) > tr((i-1)*L_s)) > tr(L_s\text{^}i/CS)/U\text{^}i$, *if* $1 \geq i$

$x(i) \in mJ_{r,r}^+$

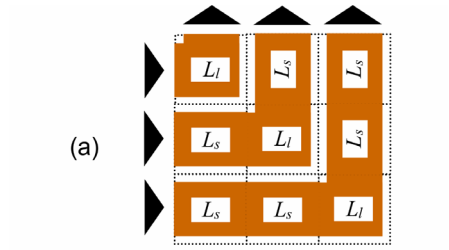
$x(1) := J_{r,r}$

$\forall i \in \mathbf{N}$: $x(i) := x(i)/((i-1)*L_s > tr(J_{r,r}^+) > tl((i-1)*L_s)) > tl(CS/L_s$

^i)/U^i , if l≥i

8.2. Layout of Metric Triple Links and Triple Forks

Metric Links and *metric Forks* are important structures for the planar layout of electronic systems, because they allow treating a bundle of parallel connections like a single one. Metric *Links* are particularly indispensable for folding a straight *Akton* structure into a desired planar layout. Each time the structure is to be expanded a metric *multiple Link* of subsort mL_s has to be inserted, and each time it is to be tilted to the left or to the right either subsort mL_l or mL_r has to be inserted. Figure 19(a) shows the structure and the program of a left-tilted strip of three chains of *Links*. Figure 19 (b) shows the structure and the program of a *multiple Fork* of three strips. Sort $F_{l,l}$ represents a unit square concealment of the term $(tl(D/L_s)/(F_{ls}>L_l))$.



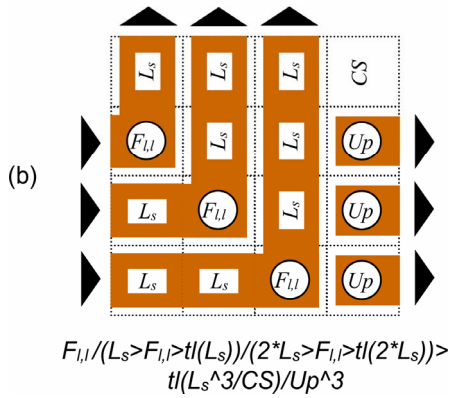


Figure 19. Two layout examples: (a) Structure and *program* of a strip of three chains turning *left*, and (b) structure and *program* of a strip of three *Forks* forking *straight* and *left*. Mind that in the second *Akton term* of *program* (b) each *Up* relates exactly to one *Down* concealed in $F_{l,l}$ according to Def. A. 21 and that also the *straight Links* L_s are metrically fixed, thus giving rise to its rectangular structure.

9. Conclusions

Every discrete physical system can be converted into a spatiotemporal network of nodes by abstracting from functions and metrics, as demonstrated in this paper. Apparently, this kind of formal description of spatiotemporal structures has not been considered up to now. Any two nodes of an abstract network are either dependent or independent. Converting a three-dimensional network into a linear one requires two homeomorphic cuts, a first one that planarizes the network and a second one that linearizes it into a sequence of symbols. Total execution of the topological program reconstructs the original spatiotemporal network. An abstract network represents an algebra with a set of fundamental sorts. These sets can be refined and concretized by subsorts, providing them with a wealth of features.

Treated this way, the spatiotemporal approach offers several surprising properties:

1) The linearization of spatiotemporal networks by means of homeomorphic twin-cuts directly generates the four letter code of DNA, which even naturally explains the skew of the double helix.

2) Extending *sort Join* of the *Akton* hierarchy by *Boolean subsorts* like *And*, *Or*, and *sort Link* by subsorts like *Wire and Not* turns the algebra into a digital circuit description language. Thus, even storage elements like *Flipflops* can be represented, *i.e.* the proper obstacle of the register transfer level calamity.

3) Reintroducing the functionality of the physical system by means of ternary interface subsorts turns the abstract programming language into a flow-controlled general data processing language. The functions may either be analog or digital. The flow-control can be restricted by partial synchronization or by total clock-control. The latter squeezes digital data processing into the word-at-time scheme of the von-Neumann-architecture, as most digital programming languages of today.

4) Reintroducing the metrics of the physical system, *i.e.* the shape and size of the components, turns abstract Akton-Algebra into a novel hardware system construction language.

5) An important property regards the layout of electronic circuitry. Every electronic circuit can always be realized by two layers only.

6) The metric algebra of *AA* can be extended to a third dimension by introducing two more *tilt functions* (see Def. M.8) for the vertical direction. It is then possible to describe devices of any shape or size by employing the dependency rules of table 1.

7) The original von-Neumann-architecture can simply be restored by pulling all stored information from the *AA*-network and collecting it in a separate main memory, and by placing a set of accumulation registers in between.

8) Aside from the considerable technical improvements of the *AA*-network-architecture the enormous augmentation in processing speed should be mentioned.

10. Acknowledgements

There are several colleagues who notably supported me during my long work towards Akton-Algebra. One of the first was Rudolf Albrecht, Innsbruck, who showed great interest in my approach when we first met in 1993 in Lessach and with whom I have had many very fruitful discussions since then. A next one to be mentioned gratefully is Walter Dosch, Lübeck, for several lucid comments. Moreover, I am most grateful to Bernd Braßel, Kiel, with whom I was in an ongoing email discussion for a couple of years. Finally, I am much obliged to Manfred Broy for some valuable hints.

REFERENCES

[1] K. A. Dill, S. B. Ozkan, M. S. Shell, T. R. Weikl, "The Protein Folding Problem," Annual Review of Biophysics,Vol. 37, 2002, pp. 289-316.

[2] M. Hazewinkel, "Homeomorphism," Encyclopedia of Mathematics, Springer, Berlin, 2001.

[3] S. Abramski, B. Coecke, "Physics from Computer Science," International Journal of Unconventional Computing, Vol. 3, No. 3, 2007, pp. 179-197.

[4] H. von Issendorff, "Algebraic Description of Physical Systems," In: R. Moreno-Diaz, B. Buchberger and J. Freire, Eds., Computer Aided Systems Theory, Lecture Notes in Computer Science, Vol. 2178, 2001, pp. 110-124.

[5] P. W. O'Hearn, H. Yang and J. C. Reynolds, "Separation and Information Hiding," *ACM* ACM Symposium on Principles of Programming Languages, 2004, pp. 268-280.

[6] B. Alberts et al.: "Molecular Biology of the Cell," Garland Publishing, New York, 2012, pp.1-46.

[7] D. Jansen et al., "The Electronic Design Automation Handbook," Kluwer Academic Publishers, Elsevier,2003,pp. 33-40.

[8] W. M. Johnston, J. R. P. Hanna and R. J.Millar, "Advances in Dataflow Programming Languages," ACM Computing Surveys, Vol. 36, No. 1, 2004, pp. 1-34.

