Life in Silico - Simulation of Complex Systems by Enzymatic Computation

Gerhard Mack and Jan Würthner
II. Institut für Theoretische Physik, Universität Hamburg

September 28, 2000

abstract We describe software and a language for quasibiological computations. Its theoretical basis is a unified theory of complex (adaptive) systems where all laws are regularities of relations between things or agents, and dynamics is made from "atomic constituents" called enzymes. The notion is abstracted from biochemistry. The software can be used to simulate physical systems as well as basic life processes. Systems can be constructed and manipulated by mouse click and there is an automatic translation of all operations into a LISP-like scripting language, so that one may compose code by mouse click.

1 Introduction

It is a fruitful idea to learn from nature how to do information processing by abstracting from what happens in the living cell.

In 1959, R. Feynman gave a visionary talk describing the possibility of building computers that were "submicroscopic". [11].

More recently, L.M. Adleman demonstrated the feasibility of carrying out computations at the molecular level by solving a standard NP-hard graph problem (similar to the travelling salesman) using molecules of DNA, and standard protocols and biochemical enzymes [1]

Here we are not interested in doing real chemistry in a bucket, but in mathematical abstractions which incorporate the basic logic of biochemical processes in the living cell and can be put on a computer. This was a dream

also of workers on Artificial Life [16]. But we may infer from Adleman's work that *life in silico* will be useful not only for simulating life processes, but for other complex problems as well.

Starting point is a philosophical principle,

> The human mind thinks about relations between things or agents.

This tells us how to model parts of the world. A unified theory of complex (adaptive) systems was built upon this principle [21]. Here we describe its implementation in software.

As in category theory [19], the relations are directed binary relations, and it is regarded as their constitutive property that they can be composed - think of *friend of a friend, brother in law, next nearest neighbor*. There are also some distinguished relations - the identities $\mathbf{1}_X$ of objects X with themselves - they play a basic role similar to the number 0 in arithmetics. Motivated by physics, a principle of locality is added by singling out some of the relations as *direct* relations, called *links*. All others are composed from them and possibly their "adjoints" - i.e. relations in the opposite direction. Links between objects form networks. Their properties are laid down in the mathematical definition of a SYSTEM, given below. Basically it makes precise the notion of *structure*.

We consider dynamics in discrete time. It is required to be local. Dynamics is also built from "atomic constituents", called enzymes. They are composed from very simple basic moves, including in particular composition of two links to a single link (friend of a friend becomes friend), making or deletion of adjoint links (reciprocation) and making or finding copies. [1]

In contrast with automata theory [30], SYSTEMs theory is supposed to be self contained. Everything that is used should be provided for by the axioms or be constructed with them. There is no other information but structure. But in the software implementation, we will not push this point of view to the extreme. We will admit numerical or text data that reside inside objects and links. Regard them as coding for structure. For more discussion on the relation between structural and numerical descriptions see ref. [21].

The whole theory may be regarded as a kind of *universal chemistry* where general objects substitute for atoms and molecules, and more general links for chemical bonds and spacial proximity.

[1] They are very simple examples of graph transformations [26]

2 Systems

According to the pioneer of general systems theory, L. van Bertalanffy [3], *a system is a set of units with relationships between them.* And according to F. Jacob [15], *every object considered in biology is a system of systems.* We precisize to [2]

Definition 1 (SYSTEM) *A* SYSTEM **S** *is a model of a part of the world as a network of* objects $X, Y, ...$ *(which represent things or agents) with* arrows $f, g, ...$ *which represent directed relations between them.*
One writes $f : X \mapsto Y$ *for a relation from a* source X *to a* target Y*.*
The arrows are characterized by axiomatic properties as follows:

1. composition. *Arrows can be composed. If* $f : X \mapsto Y$ *and* $g : Y \mapsto Z$ *are arrows, then the arrow*

$$g \circ f : X \mapsto Z$$

is defined. The composition is associative, i.e. $(h \circ g) \circ f = h \circ (g \circ f)$*.*

2. adjoint. *To every arrow* $f : X \mapsto Y$ *there is a unique arrow* $f^* : Y \mapsto X$ *in the opposite direction, called the adjoint of* f*.* $f^{**} = f$ *and* $(g \circ f)^* = f^* \circ g^*$*.*

3. identity. *To every object* X *there is a unique arrow* $\mathbf{1}_X : X \mapsto X$ *which represents the identity of a thing or agent with itself.*

$$\mathbf{1}_X = \mathbf{1}_X^*, \qquad \textit{and} \qquad \mathbf{1}_Y \circ f = f = f \circ \mathbf{1}_X$$

for every arrow $f : X \mapsto Y$*.*

4. locality*: Some of the arrows are declared* direct *(or fundamental); they are called* links*. All arrows* f *can be made from links by composition and adjunction,* $f = b_n \circ ... \circ b_1, (n \geq 0)$ *where* b_i *are links or adjoints of links; the empty product* $(n = 0)$ *represents the identity.*

5. composites: *The objects* X *are either* atomic *or* SYSTEMs*. In the latter case,* X *is said to have internal structure, and the objects of the* SYSTEM X *are called its* constituents*.*

[2] Axioms 1,3 are those of a category [19]. Therefore SYSTEMs are categories with a $*$-operation and a notion of locality

6. non-selfinclusion: *A* SYSTEM *cannot be its own object or constituent of an object etc. Ultimately, constituents of ... of constituents are atomic.*

A Semiadditive SYSTEM *satisfies the same axioms, except that arrows may be composed from links and their adjoints with the help of two operations,* $\circ$ *and* $\oplus$*. The* $\oplus$*-operation adds parallel arrows. It makes the set* $\mathbf{S}(X, Y)$ *of all arrows with given source* X *and target* Y *into an additive (=commutative) semigroup. The distributive law holds*

$$(f_1 \oplus f_2) \circ (g_1 \oplus g_2) = f_1 \circ g_1 \oplus f_2 \circ g_1 \oplus f_1 \circ g_2 \oplus f_2 \circ g_2$$

An arrow o *is a zero arrow if* $f \oplus o = f$ *for all* f*. It is understood that arrows are modulo zero arrows.*

The rationale behind the $\oplus$ operation is that it should be possible to interpret two parallel links as a single link, and similarly for arrows.

Remark 1 *In a Semiadditive* SYSTEM*, the set* $\mathbf{S}(X, Y)$ *can be extended to an additive group if and only if* $f \oplus h = g \oplus h$ *implies* $f = g$*, whatever is* h*.*

In the following, we will sometimes refer to composition with $\circ$ as *multiplication*, and to $\oplus$ as *addition* of links.

In addition to the axioms, it is assumed that constituents, constituents of constituents etc. of objects of $\mathbf{S}$ are not objects of $\mathbf{S}$. This assumption is subject to being weakened, but weakening it may require adjustment or redesign of software. It would be interesting to weaken also the axiomatic property of non-selfinclusion. But this is a very subtle operation, cp. the discussion in section 8.1

We found it convenient to generalize the axiomatic notion of identity arrow by introducing *identity links* as a special type of link which may connect objects that are identical in the sense of indistinguishable.

Definition 2 (Subsystems) *A* subsystem $\mathbf{S}_1$ *of a* SYSTEM $\mathbf{S}$ *is generated by a set of objects in* $\mathbf{S}$ *and a set of links in* $\mathbf{S}$ *between these objects. Its arrows are all arrows in* $\mathbf{S}$ *that can be composed from these links and their adjoints.*

The boundary *of* $\mathbf{S}_1$ *consists of the links in* $\mathbf{S}$ *with target in* $\mathbf{S}_1$ *which are not links or adjoints of links in* $\mathbf{S}_1$*.*

The environment *of* $\mathbf{S}_1$ *is the system generated by the objects of* $\mathbf{S}$ *not in* $\mathbf{S}_1$ *and the links between them.*

In the software, subsystems can be specified by marks in two different ways, either by marking the (inward) links in its boundary, or by marking the links which belong to it.

In the description of the software design, we use the following

Nomenclature 1

Valence *The links with target X are called the* Valences *of X*

Radical *An object X together with all its valences is called a* Radical.*Given some link b, the radical which contains its source (object) is called the* source radical *of b.*

Membrane *Marks on links which identify the boundary of a subsystem are called* Membranes

Path *A* Path *from X to Y is a sequence $b_1, ..., b_n$, where b_i are links or adjoints of links, X is the source of b_1, Y is the target of b_n, and the target of b_i is the source of b_{i+1} for $i = 1, ..., n-1$. The empty path ($n = 0$) from X to X is identified with $\mathbf{1}_X$. Paths are special subsystems. Marks on links which belong to a certain subsystem are called* **path-links**.

In simulations of biological organisms and their parts, membranes can be used as models of cell membranes or envelopes of organs. Path-links are used to model blood vessels and other transport channels, and neural nets. Attachig enzymes can convert terminal objects of neural nets to sensors and effectors, and objects within blood vessels to pumps. And copy processes can make organisms grow. Any SYSTEM with all its interna can be copied by purely local processes, induced e.g. by propagating shocks, cp. section 8.1.

2.1 Basic equations

In SYSTEMs theory, basic equations are often of the form $l = \mathbf{1}_X$, where $l : X \mapsto X$ is the arrow specified by some path. Maxwell's equations have this form [21].

Validity of constraints of this form can distinguish between modes of being [21]. For instance, space time is distiguished by having a Lorentzian geometry, [3] and material bodies are in space and obey conservation laws.

[3]Differential geometry is a father of SYSTEMs theory, with parallel transporters as arrows

2.2 Isomorphic systems

The notion of isomorphic systems is somwhat subtle. Basically, two SYSTEMs $\mathbf{S}^1$ and $\mathbf{S}^2$ are isomorphic if there exists a structure preserving map $F : \mathbf{S}^1 \mapsto \mathbf{S}^2$ which maps objects into objects and links into links, and whose inverse exists and has the same property. This induces a map of arrows f into arrows $F(f)$. We require of a structure preserving map F that source and target of $F(f)$ are the images of source and target of f, and [4]

$$\begin{aligned} F(f \circ g) &= F(f) \circ F(g), & (1)\\ F(\mathbf{1}_X) &= \mathbf{1}_{F(X)} \ . & (2)\\ F(f^*) &= F(f)^* & (3) \end{aligned}$$

There are also anti-isomorphisms. They relate complementary shapes. For an antiisomorphism, eq.(1) is replaced by

$$F(f \circ g) = F(g) \circ F(f), \tag{4}$$

and source and target are interchanged.

It is NOT required that the internal structure of corresponding objects matches when they are not atomic. Nonatomic objects are regarded as black boxes. The internal structure of black boxes (nonatomic objects) is declared irrelevant when one does not distinguish isomorphic SYSTEMs, and so is the distinction between atomic and nonatomic objects. The only usage of the internal structure is in constructing links of the SYSTEM and their composition $\circ$. One does not look into black boxes anymore once they are in place and connected.

We speak of a *strong isomorphism* if it IS demanded that corresponding nonatomic objects are strongly isomorphic SYSTEMs. (This recursive definition makes sense because of the axiom of non-selfinclusion.)

3 Dynamical systems

We consider dynamics in discrete time. A deterministic dynamics of first order shall determine a SYSTEM $\mathbf{S}_{t+1}$ at time $t+1$ and links between objects

[4] These are the axiomatic properties of functors of a category [19], with the added demand that the map is local in the sense that it maps links to links, and preserves the $*$-operation. Anti-isomorphisms are invertible contravariant functors

X in $\mathbf{S}_{t+1}$ and objects Y in $\mathbf{S}_t$ from a given a SYSTEM $\mathbf{S}_t$ at time t. If there is a link between $X \in \mathbf{S}_{t+1}$ and $Y \in \mathbf{S}_t$, X is said to be descendent of Y, and Y is ancestor of X.

It is required that

1. Every object $X \in \mathbf{S}_{t+1}$ is descendent of at least one object Y. Conversely, $\{Y_1, ..., Y_n\}$ can only be the set of ancestors of some X if the system generated by $Y_1, ..., Y_n$ and identity links between them is connected.

   If $(n > 1)$ we say that there is fusion. An object may have more than one descendent (e.g. copies or translations).

2. The dynamics is local in the sense that the isomorphism class (resp. strong isomorphism class) of a subsystem of $\mathbf{S}_{t+1}$ depends only on the SYSTEM generated by the objects in a 1-neighborhood of its ancestors and the links between them.

A deterministic dynamics determines $\mathbf{S}_t$ up to isomorphism, a strongly deterministic dynamics determines it up to strong isomorphism, i.e. including the internal structure of nonatomic objects. A dynamics is called universal if it is well defined for every SYSTEM $\mathbf{S}_t$ whatever.

The idea of enzymatic computation is

first, to build dynamics from "atomic constituents" (elementary moves) which we call enzymes.

second, to incorporate the information about the dynamics into the initial state by attaching enzymes to links and objects of $\mathbf{S}_t$.

The enzymes may act conditionally. To this end they are combined with predicates which examine the structure of a neigborhood in the system. A predicate `p` determines a boolean function `p.evaluate(r,v)` where `r` is a radical, and `v` is (pointer to) a valence, typically of `r`. The pair (predicate, enzyme) is also called a *mechanism*. The enzymes are abstractions not only of the biochemical enzymes [2] which govern life processes in cells, but of any agent of change. Biochemical enzymes show specificity [2], i.e. they come with their predicates inseparably build in. We shall occasionally speak of enzymes when we really mean mechanisms.

There are several kinds of basic enzymes, i.e. elementary moves.

motion An arrow becomes promoted to link. Either

a) The composite of two (or more) links becomes a link, i.e. indirect relations become direct. *example 1: Friend of a friend becomes friend. example 2: Equations of motion of fundamental physics like Maxwells equations [21, 22] or motion of particles in space.[Transport of material involves motion of particles in space (or an effective decription of that in terms of algebraic values)]. example 3: catalysis, figure 1. example 4: logical deductions, cp. ref.[21, 27].*

b) The adjoint of a link which was not a link becomes a link. *example: reciprocation*

The converse - adjoint of a link loses the status of a link - is also subsumed under motion. Motion is reversible.

growth An object has two or more descendents - e.g copies which are made or taken from the environment. Fusion of objects is also subsumed under growth, and so is the making of an object with internal structure from a subsystem. Growth is reversible if descendents are connected by identity links (cp. section 2 after definition 1). Removal of the identity link would be subsumed under death, s. below.

death An object may disappear with its links, links may disappear together with their adjoints. Death is usually irreversible.

cognition A new link is made between objects X_1, X_2 with matching internal structure. Locality requires that X_1, X_2 are connected by a path of short length in $\mathbf{S}_t$, i.e. there is a preexisting relation between them. But the new link need not be the arrow determined by the preexisting path. *example: a chemical bond may be established when there was spacial proximity before.* Links made this way are called *cognitive links*.

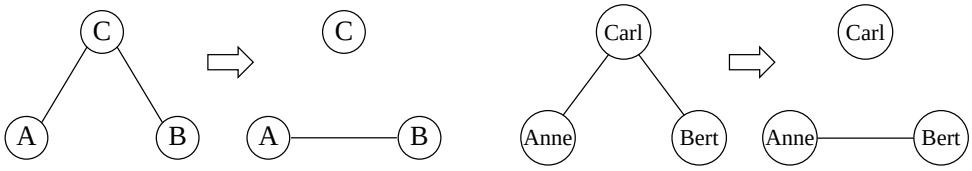


Figure 1: Catalysis in chemistry and elsewhere. A catalyst `C` binds molecules `A` and `B`. First a substrate-enzyme complex is built, where `A` and `B` bind to `C`. Next the composite arrow from `A` to `B` becomes fundamental.

enzyme management Enzymes are attached to objects and links of $\mathbf{S}_{t+1}$, or their action somewhere at time $t+1$ is specified in other ways.

The making of cognitive links only makes sense if the dynamics determines future SYSTEMs up to *strong* isomorphism, because the internal structure of nonatomic objects matters. Cognitive links can be of several types f. f is a boolean function ("matching structure") of two objects, typically determined by a predicate p with the property that $p(r, v)$ depends only on the object of radical r and the source object of valence v. Its return value is also supposed to be obtained by enzymatic computation (with access to the internal structure of objects which are SYSTEMs).

The most important example of matching structure are isomorphisms and antiisomorphism. The latter implement the *lock key* mechanism which is responsible for specificity in biochemistry [2]. Determining isomorphisms of graphs is an NP-hard problem. But this is not very relevant here because in practise one will encounter graphs with special properties. Matching structures can also be detected by neural nets, here implemented as subsystems made with path-links.

We will subsume the making of identity links between indistinguishable atomic objects under cognition as well, and regard such identity links as prototypical examples of cognitive links.

A description of the most important enzymes will be given later.

So far we talked about deterministic dynamics. In a stochastic dynamics, enzymes act with certain probabilities.

3.1 Concurrency

The action of enzymes at different locations will not commute in general. This results in what is known to computer scientists working on parallel computing as the *concurrency problem*. Petri nets are a well know example [25]. In the spirit of biology, one may just ignore this problem, admitting some randomness or indeterminacy in the dynamics, e.g. through a hidden dependence on an ordering of the valences of a radical. But it is useful to have the option of specifying a well defined deterministic dynamics for SYSTEMs.

We propose to achieve this by a generalization of Jacobi sweeps.

Let us first recall the notion of a Jacobi sweep. Consider a grid made of nodes (objects) connected by links, where the nodes and/or the links carry some data. Given a cost function whose arguments are the aforementioned

data and which is a sum of contributions which depend only on a neighborhood of individual nodes or links. One may attempt its minimization by relaxation. One makes sweeps through the grid, updating data at individual nodes or links such that the cost function is minimized under the constraint that all other data remain frozen. In a Jacobi sweep (as opposed to Gauss Seidel sweeps)[13], the data after sweep $t+1$ are determined solely by the data after sweep t. As a result it does not matter in which order the nodes and links of the grid are visited - the effect of the individual updating operations will commute.

We may proceed in the same way, except that we divide the making of the SYSTEM $\mathbf{S}_{t+1}$ at time t from the SYSTEM $\mathbf{S}_t$ at time t into several steps which take place at times $t+\frac{1}{4}$, $t+\frac{1}{2}$, $t+\frac{3}{4}$ and $t+1$. In intermediate steps, there will be links connecting objects in $\mathbf{S}_t$ with objects in S_t+1. We call them "time links"

We may divide enzymes into classes according to whether they act at time $t+\frac{1}{4}$, $t+\frac{1}{2}$ or $t+1$. The intermediate step at time $t+\frac{3}{4}$ was introduced for pedagogical reasons only. The enzymes which act at time $t+\frac{1}{4}$ will be called "object-making", those at time $t+\frac{1}{2}$ "link-making". The link-making enzymes may also put marks on newly made links to indicate information on their adjoints.

As explained in section 3, our processes are such that every object Y in $\mathbf{S}_{t+1}$ is descendent of (at least) one object X in $\mathbf{S}_t$.

Ignoring the attachment of enzymes to $\mathbf{S}_{t+1}$ at first, the course of events is like this (cp. figure 2 for an example) .

$t+\frac{1}{4}$ Make descendents $Y_1, ..., Y_n \in \mathbf{S}_t$ of *objects* $X \in \mathbf{S}_t$, if any; $(n \leq 2)$. Connect them by time links. When $(n > 1)$ make identity links between identical copies $Y_1, ..., Y_n$ if demanded by an enzyme.

$t+\frac{1}{2}$ Make *links* that will end up in $\mathbf{S}_{t+1}$. Every link-making enzyme may contribute some link or links, in a manner which depends on its neighborhood in $\mathbf{S}_t$. For pedagogical reasons, the links are made at time $t+\frac{1}{2}$ to connect one object in $\mathbf{S}_t$ and one object in $\mathbf{S}_{t+1}$, and at time

$t+\frac{3}{4}$ the ends in $\mathbf{S}_t$ of such links are lifted to $\mathbf{S}_{t+1}$ by composition with a time link.

$t+1$ Depending on the marks on the newly made links in $\mathbf{S}_{t+1}$, adjointness relations are established among links, or new fundamental adjoints are

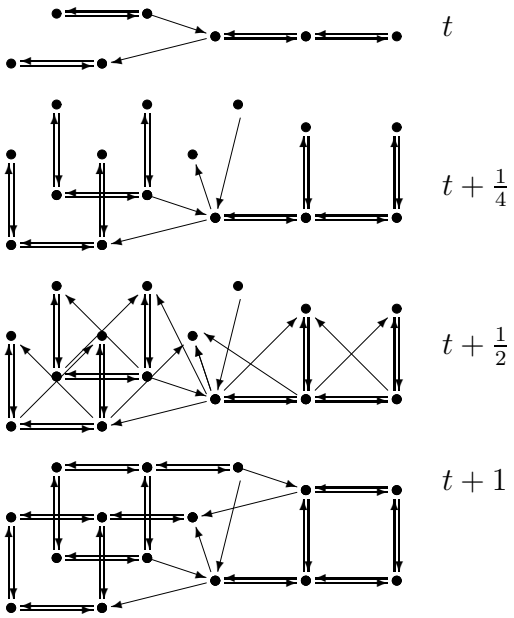


Figure 2: splitfork dynamics, concurrent version

made. The operation is local in that it involves only the individual pairs of sets $\mathbf{S}(Y, Y')$ and $\mathbf{S}(Y', Y)$ of links between given objects Y, Y'.

So far we neglected to say how the enzymes are put into the SYSTEM $\mathbf{S}_{t+1}$. Object making enzymes are attached to descendents or identity links between them in step $t + \frac{1}{4}$. Link-making enzymes are attached to links in step $t + \frac{1}{2}$.

Dynamics of this kind is well defined because all operations within anyone of the steps commute.

In the end, the time links may be removed (if desired). This step is not shown in figure 2.

Unfortunately the use of Jacobi sweeps is less convenient than admitting a hidden dependence of the order in which enzymes at neighbouring locations act on an ordering of valences in radicals. This is so because the composition of chains of enzymes which act one after another is not a simple matter anymore.

4 The System Class Laboratory

The System Class Laboratory (SyCL) is a software package composed of three parts

core package is a C^{++} class library which encodes the basic functionality demanded by the axioms of a *semi-additive system* and enzymatic dynamics on it.

presentation package includes a graphic interface. Using it one can construct and manipulate such SYSTEMs by mouse-click, invoking the conditional action of basic enzymes and of compound enzymes made from them. The presentation package also includes to translate SYSTEMs to and from **XML**, i.e. text.

interpreter and parser package is based on a LISP-like scripting language. One can write programs in this language and run them. Alternatively, operations carried out by mouse click on the graphic interface are automatically recorded as commands of the scripting language. In this way one may compose programs by mouse click.

5 The class library

5.1 Basic Data Types

In principle, SYSTEM theory is self-contained [5]. There are no data in SYSTEMs other than their structure, and no states of any part of a SYSTEM other than its structure. The miracle is how much can be modeled with so little building material.

In practice, it is nevertheless convenient to admit data inside objects and links which can be thought to code for internal structure in some way. (Links with internal structure can be thought of as SYSTEM's with two interfaces)

The data inside links are instances of a class *AlgebraicValue*. They admit algebraic operations. They can be added, multiplied and multiplied with real numbers in a manner which is described below in subsection 5.2. Their equality can also be ascertained. The class *AlgebraicValue* is derived from a

[5] in contrast with automata theory [30]

class *Value* which does not have the algebraic operations as methods. Objects can contain arbitrary *Value*s. Multiplication and addition of algebraic values that reside inside links is invoked when links are composed using the axiomatic $\circ$ and $\oplus$-operations. Several different classes are derived from AlgebraicValue, including the class Predicate. We mention it separately because of its important role. Enzymes may also be attached to objects and links in the guise of EnzymaticValue, also derived from AlgebraicValue.

The other data types correspond with Nomenclature 1, except for the following implementational detail. The adjoint of a link b with source X which is not itself a link is nevertheless included in the list of Valences of the radical containing X, and is marked as a "virtual valence". This is technically convenient because the target of b can be addressed as source of its adjoint.

The basic data types are

Value, AlgebraicValue contain text or numerical data with operations as described

Objects may contain a singly linked list of *Values*. They can be copied.

Valences contain a pointer to their *source radical*, a pointer to their *adjoint*s, and possibly: a singly linked list of *AlgebraicValue*'s, a singly linked list of *Membrane*s, a singly linked list of *Path*s.

By definition, the adjoint is a valence of the source radical, possibly virtual.

Composition $\circ$ of a valence with valences of its source radical is defined.

Parallel valences may be added to a single valence.

Radicals contain an object, a doubly linked list of valences, and possibly a position in 3-dimensional space.

Predicates determine a boolean function which takes as argument a pair `(r,v)`, where `v` is a pointer to valence (maybe NULL) and `r` is a radical (typically the target of `v`). Predicates can be composed with junctors "and, or, not". Among the predicates are keys which permit to pass membranes selectively.

Membranes are attached to valences. They mark the boundary of a subsystem. They carry a string or a `void*` pointer as code to distinguish them.

PathLinks contain a reference to a valence and data to identify the precursor, successor and adjoint pathLink. A valence knows the pathLinks that pass through it, cp. above.

System A connected nonempty SYSTEM is identified by a pointer `root` to one of its radicals, and possibly by the code of a membrane which bounds it.

From `root` one can access the source radicals of its valences, and so on. All the information on the SYSTEM is in the radicals which can be accessed in this way.

Enzyme Enzymes have a method `Radical & act(r,v)` which takes as its arguments a radical `r` and a pointer `v` to a pointer *v to a valence (of `r`). `*v` may be NULL. The action is on a neighborhood of `r`. `*v` may be changed to point to a different valence, therefore a pair `(Radical &, Valence *)` is effectively returned, which can be used for input to another enzyme.

One may define a *path distance* d between objects X and Y by use of paths $\{b_1, ..., b_n\}$. d is the minimal value of n among all the paths from X to Y.

5.2 AlgebraicValues

The implementation of algebraic operations with AlgebraicValues makes essential use of object oriented design features like inheritance and virtuality. [12]. We assume that the reader is familiar with these.

The algebraic operations satisfy the usual laws of associativity and distributivity. Multiplication may be noncommutative, and in one special case (the max-plus-"algebra"), subtraction is undefined.

There is an inheritance tree which specifies how different classes of algebraic values are derived from the base class *AlgebraicValue*. Each class has its (virtual) methods for multiplication with members of the same class from the right or left, for addition, and for multiplication with real numbers α. In some cases (like strings) the multiplication with real α does nothing if $\alpha \neq 0$, and converts to a zero-element otherwise. Given any two classes A, B of algebraic values, there is a unique class C, the " least common ancestor", such that A and B are derived from C, but not from any class that is derived from C. The result of right multiplication $*=$ is computed by invoking the

multiplication-method of C. Addition and left multiplication are treated in the same way.

The inheritance tree is shown in figure 3.

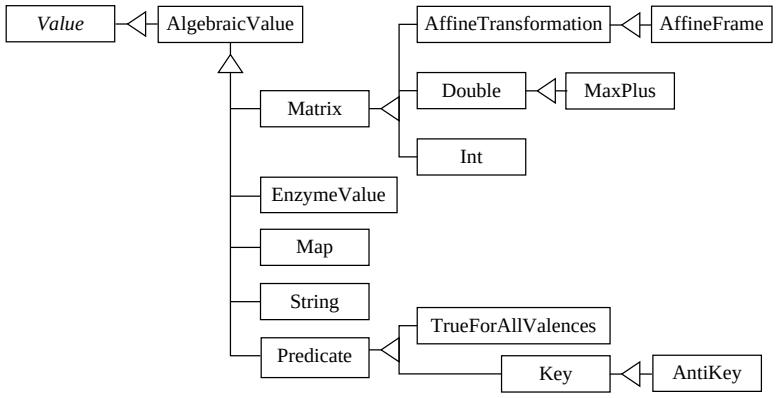


Figure 3: The class diagram for different types of values.

There is one type of algebraic values of interest where the condition of remark 1 is violated, the *max-plus* or *timetable* "algebra". Its data are real numbers (or real matrices), $a \circ b = a + b$ (addition of real numbers) and $a \oplus b = max(a, b)$. For matrices, the operations are entry-wise. This "algebra" is useful in so called *discrete event systems* [4], and in optimization problems where one requires the maximum of errors which are given by real return values of local cost functions.

There may be a (singly linked) list of AlgebraicValues in a link. In this case, multiplication is element by element in the list.

Values in Objects need not be of type *AlgebraicValue*. But for instances η of useful data types, multiplication with AlgebraicValue A from the left is defined and is associative (i.e. $(A * B) * \eta = A * (B * \eta)$). The multiplication may be trivial, i.e. $A * \eta = \eta$ for all A. In case η is itself of type AlgebraicValue, multiplication is defined as described above.

Definition 3 *(*Parallel Transport*) Given a link b with AlgebraicValue A and the value η of its source object Y, $A * \eta$ is regarded as a potential value of the target object X of the link b, and the operation is called the* parallel transport *of η from Y to X* along the link b.

The parallel transport along links may be composed to parallel transport along paths $\{b_0, ..., b_n\}$. Parallel transport along $\mathbf{1}_Y$ is the trivial map $\eta \mapsto \eta$. The notion comes from lattice gauge theory of elementary particles [6] and general relativity.

Remark 2 *Because of associativity, the result of the parallel transport depends actually only on the arrow defined by the path.*

It may happen that the result of parallel transport of some type of value from Y to X is independent of the chosen path from Y to X. In this case the type of value is called an *invariant*.

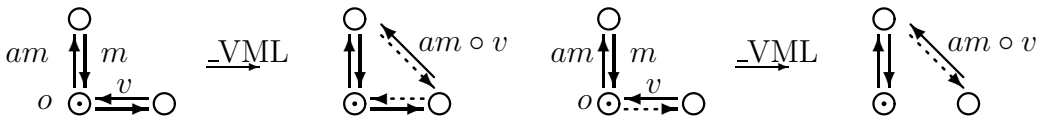
Invariants have a global meaning in a connected SYSTEM, because they may be parallel transported from any Y to any X in a unique way. Important examples are i) quantities of any chemical substance which is transferred from object to object by diffusion, ii) payments in an economy.

Counterexamples are any kind of "non-ideal communication", such as communication of humans in natural language. Utterances in natural language do not generally have a globally well defined meaning - one's owl is the other's nightingale. Gossip is an example. When the message comes back to speaker X who send it out, it may have undergone dramatic change. So the result of communication around some loop is not the same as along the identity $\mathbf{1}_X$.

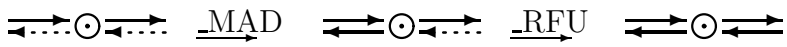
5.3 Enzymes

We proceed to describe the most important micro-enzymes, beginning with

motion: It involves either multiplication of two links, or promotion of a virtual adjoint of a link to the status of a link. We have enzymes $\_VML$ for left multiplication and $\_AMR$ for right multiplication. Given a $\_VML$-enzyme `e` with attached predicate `p`, `e.act(r,m)` will go through all valences `v` of `r`, and multiplies those `v` from the left with the adjoint of `m` for which `p.evaluate(r,v)` returns TRUE. An illustration is found below. In this figure, virtual valences are indicated as dotted lines. The $\_AMR$-enzyme is similar.



Alternatively, motion may involve the promotion of the adjoint of a link to the status of link. There are enzymes to do that, one of them is the $\_MAD$-enzyme. It makes adjoints of all valences into links which fulfill the condition set by a predicate. $\_RFU$ acts in the same way, except that the argument `v` is replaced by its adjoint, and the source of `v` substitutes for `r`.



To change a valences status from fundamental to virtual, an enzyme $\_MVV$ is used.

Growth: The most important enzyme is $\_CPO$ which copies an object, and links original and copy by an identity link. (Another version searches for the copy in the environment). The object may be composite, i.e. a SYSTEM. $\_CPO$ appears in the first micro-enzyme in the splitFork enzyme as shown in figure 6 in section 8.1. In the figure there, some predicates are indicated by a string-code {?...} enclosed in braces. Their meaning is as follows

?nLK `v` is not a link (i.e. is a virtual valence)

?iAD adjoint of `v` is a link, negation ?nAD.

?iID `v` is an identity link.

?hLS detect a triangular structure.

There are also enzymes like $\_SYA$ which make objects of subsystems whose boundary is marked by membranes.

Death: links are removed by $\_RTV$ or $\_RMV$ enzymes which kill the particular valence given as argument, or all valences which fulfill a condition. (The $\_RMV$-enzyme operates in figure 6 to remove identity links.)

A removal of a radical with all its valences and their adjoints by the $\_DEL$-enzyme.

Enzyme management enzymes: Among them there are presentation enzymes. They have relatives in cell biology [2]. They hand arguments (r,v) to another enzyme. For instance, for a $\_PRS$-enzyme `e`, ("present Source") `e.act(r,v)`, presents `(s, adjoint(v))`, (`s`=source of `v`) to be acted on by the next enzyme following `e` in an enzyme chain. Predicates may make the presentation conditional.

There are also quantors such as `fALL` which invokes the action `e.act(r,v)` of an enzyme `e` for all valences `v` of a radical `r` which fulfill a condition; `e` is appended as tail of an enzyme chain with head `fALL`. (A *forAll* Quantor exists also for predicates.)

Unless one uses the concurrent version of the dynamics (cf. section 3.1), the present implementation for single processor machines offers the option of reporting an enzyme to an agenda together with the argument pair `(r,v)` it will act on, instead of attaching it to the valence `v` or the object of `r` in the SYSTEM. There are enzymes to do that. A *_Sag*-enzyme ("source-to-agenda") in an enzyme chain reports the whole chain together with `(s, NULL)`, `s` the source of its `v`-argument.

Membranes: The "push membrane" enzymes *_PIM* and *_POM* surrounds a radical by a membrane (attached to all its valences) or pushes the radical inside resp. outside a membrane that goes through one of its valences resp. its adjoint. The membrane's code is specified by a predicate.

Paths: There are enzymes to make or prolong paths along specified valences.

6 Graphic Interface

There is a graphic interface which can be used to construct SYSTEMs and manipulate them by use of enzymes. All this is done by mouse click, except that numerical or text-data need to be entered from the key board when they are not copied from somewhere.

All operations are recorded and translated into code by use of the LISP-like scripting language which is described in the next section. In this way, programs which construct SYSTEMs or code for processes on SYSTEMs can be composed by mousclick.

In detail one may
- Position objects or delete them
- Link them by valences (with or without adjoint link) or delete links.
- Enter data (*Value*s) into objects or links.
- Select basic enzymes and accompanying predicates from lists.
- Build composite enzymes as chains of basic "micro-enzymes".
- Invoke conditional application of enzymes to radicals `r` with or without a selected valence `v`.

Among the enzymes there are some which
- create a membrane around an object, or push a membrane beyond other objects.
- create paths or prolong them
- create copies of individual objects, of the whole system, or of a subsystem

bounded by a membrane.
- induce sweeps through the whole SYSTEM [or through a part of it which is bounded by a membrane or marked by path-Links], applying the tail of an $aFRK$-enzyme to every radical in it once.
- induce sweeps which are reflected at the locus of maximal path distance, assemble data from all the objects and links of the system and report them to the SYSTEM' root. The computation of maxima of real data at all objects is a sample application.
- induce diffusion within the SYSTEM or a subsystem.

Coordinates are assigned to each object. A panel allows to rotate and translate the coordinate system. Different windows admit different views on a system and its dynamical behavior.

6.1 Permanence: XML

Methods are provided to translate a SYSTEM into **XML** and conversely to construct a SYSTEM from **XML**-code. The methods can be invoked through the graphic interface.

In this subsection we assume the reader is familiar with **XML**. An introduction to **XML** can be found in [18].

The dtd (*document type definition*) listed in [31] defines tags and their possible attributes for the systems elements. An internal id number is assigned to each object and valence to let a valence remember its source, target and adjoint valence.

The `<system>` tag encloses anything else inside the document. The `<object>` tag carries an objects id number and a position as attributes. `<value>` tags may be enclosed. The `<valence>` tag carries a valences id number, the id numbers of its source and target objects and of its adjoint valence as attributes. Again, `<value>` tags may be enclosed. The `<value>` tag carries the value type and the value content as attributes.

The way XML is designed it allows future extensions while assuring up- and downward compatibility.

7 The Scripting Language

If a particular kind of complex problem occurs very often, it might be worthwhile to express instances of the problem as sentences in a simple language[12]. Solving a problem then means to interpret the sentence, for which an *interpreter* is needed.
Based on a certain grammar which defines a simple language, the interpreter represents and interprets sentences in that language. To be efficient, character strings which make up the sentences are transformed into trees of objects, which are easy to handle by the interpreter. This transformation is done by a software called *parser*.
The grammar used for our purposes is very similar to the one used by the *list processing language LISP*.

```
expression ::= list | atom
list ::= ‘(’ expression* ‘)’
atom ::= ‘a’ | ‘b’ | ... | ‘z’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘$’ |
             { ‘a’ | ‘b’ | ... | ‘z’ }* | ‘ quote ’ expression
```

In object oriented software design, a well known interpreter pattern (suggested in ref.[12]) has shown to work fine in most cases. To implement a parser and interpreter for the SyCL program package, this pattern is used in combination with a composite pattern [12] and extended by one more abstraction in the way shown in figure 4.

The ***Element*** class is abstract. It provides polymorphism by assuring that every object is of type *Element*. A Element can thus either be an object of type **List**, which consists of further Elements, or an object of type ***Atom***, where *Atom* is abstract again. The following classes are derived from class *Atom*.

A list can consist of elements of any kind. To evaluate a list, the first element has to be a function, or an enzyme, where the following elements serve as parameters, which have to be evaluable unless they are quoted (i.e. they start with a quote).

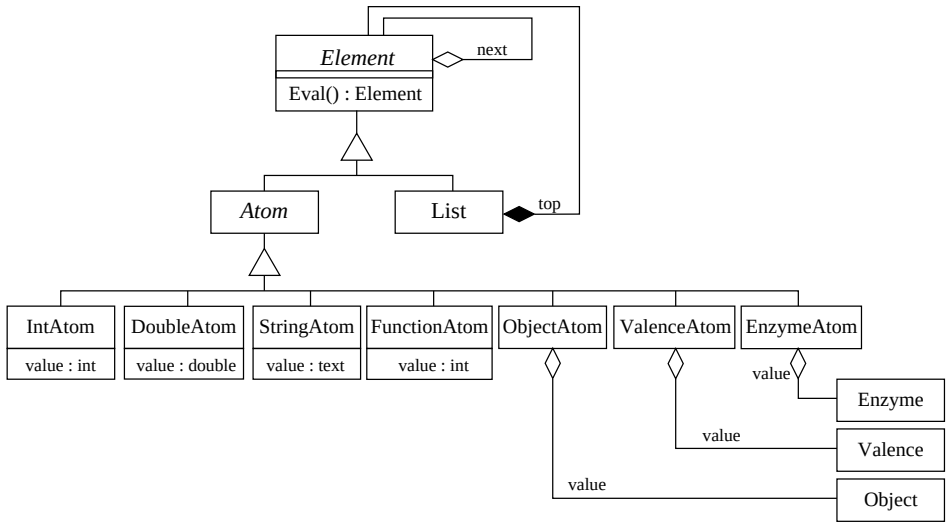


Figure 4: The class diagram for the SyCL parser.

Implemented Functions

The commands, this interpreter can execute are listed below. Enzymes can be applied as well, taking a radical or a list of a radical and a valence as a parameter.

(adjoint v) returns the adjoint valence to valence v
(append expr li) appends the expression *expr* to the list *li* and returns this list. The original list remains unchanged!
(car li) returns the first element of the list *li*
(cdr li) returns the list *li* without its first element
(connectto a x y z) searches the system for an object at the position (x, y, z) and assignes the variable a to it.
(delete a) removes element a from the variable list and deletes it (compare *remove*).
(edit a b) changes the value of an object or valence a to the SyCL value b.
(enzyme expr) defines a new enzyme from the expression *expr*. *expr* has to be of the form *xxxx=enz1enz2enz3....* and must be quoted in order not to be executed as a command! The newly defined enzyme is automatically inserted into the enzymelist as well as the enzyme menu from the GUI.

(enzymes) returns the list of enzymes.
(eval expr) evaluates the expression *expr*.
(equal a b) or (eq a b) returns 1 if a and b are equal, an empty list otherwise.
(for (n i0 i1) (expr1 expr2 expr 3 ...)) evaluates the expressions *expr1, expr2* and *expr3 ...* with the variable n taking integer values from *i0* to *i1*
(funlist) returns the list of implemented functions.
(greaterthan a b) or (gth a b) returns 1 if a is greater than b, an empty list otherwise.
(if expr (expr1 expr2 expr3 ...)) evaluates the expressions *expr1, expr2* and *expr3 ...* if *expr* is true.
(length li) returns the number of elements of the list *list*.
(lessthan a b) or (lth a b) returns 1 if a is less than b, an empty list otherwise.
(list a b ...) returns a list with the parameters *a, b, ...* als elements.
(load filename) loads and evaluates commands in a file *filename*. This can be done more comforably with a file-select box via the GUI *open file* menu entry.
(minus a b) (- a b) returns the difference of a and it b.
(newo a) creates a new SyCL object of SyCL value a and returns an object element.
(notequal a b) or (neq a b) returns an empty list if a and b are equal, 1 otherwise.
(nth n li) returns the nth element of a list *li*. Counting starts at zero!
(nthval n a) returns the nth valence of the valences directed toward the object a (counting from 0).
(oblist) returns the list of variables.
(poso a x y z) repositions the object designated by variable a at (x, y, z).
(plus a b) (+ a b) returns the sum of a and b.
(rand n) returns an integer random number between zero (inclusive) and n (exclusive).
(remove a) removes element a from variable list (compare *delete*).
(replacenth n li expr) replaces the nth element of a list *li* by *expr* and returns this list. The original list *li* remains unchanged!
(run '(e v)) lets the enzyme e act on the source of valence v.
(set a b) same as **setq**, but evaluates a first.
(setl l v a b) a and b have to be object elements. An unidirectional valence

is created between *a* and *b* with a SyCL values *v*. The valence element is assigned to the variable *l*.
(seto a ['int | 'real | 'bool | 'enz] ['vector m | 'covector n | 'matrix m n] b... [x [y [z]]]) creates a SyCL object with a SyCL value *b* (evaluated) and assigns the object element to the variable *a*. Creating a vector of dimension m, a covector of dimension n, or a matrix of dimension *m*x*n* requires *m*x*n* elements *b*! *x, y, z* can be appended to specify coordinates.
(setq a b) creates an expression, sets it to *b* (evaluated) and assigns it to the variable *a*.
(setv v v0 v1 a b) *a* and *b* have to be object elements. A bidirectional valence is created between *a* and *b* with SyCL values *v0* and *v1* for the valence and its adjoint. The valence element is assigned to the variable *v*.
(times a b) or (* a b) returns the product of *a* and *b*.
(valences a) returns the number of valences directed toward an object *a*.

The following script may serve as an example of motion along a valence *v* between two objects *a* and *c*:

```
(seto a 1)
(seto b 2 100)
(seto c 3 0 100)
(setv v 1 1 a b)
(setq w (adjoint v))
(setv m 1 1 a c)
(setq q (adjoint m))
(enzyme 'move=_AMR_RMV{?nAD}_PRS_RFU)

(move '(a m))
```

The last command invokes the action of the above defined enzyme `move` with object *a* and valence *m* as parameters.

8 Some basic Processes

8.1 Enzymatic Copying

Let us call a link *bidirectonal* if its adjoint is also a link, *unidirectional* otherwise. Note that the terms are used to characterize individual links, not pairs (link, adjoint link).

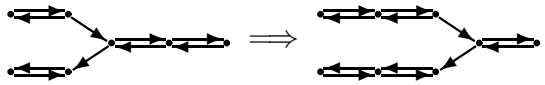


Figure 5: Replication fork dynamics.

There exists an enzyme, call it the splitFork-enzyme, whose continued conditional application to a finite SYSTEM **S** whose links are all bidirectional, produces after some finite time two copies of **S**.

This is a generalization of the asymmetrical replication fork mechanism which the living cell uses to copy DNA [2]. It works not only for chains of pairs of bidirectional links (which mimick the double helix of DNA), but for any SYSTEM whose links are all bidirectional, independent of its topology. [All the interna like (possibly overlapping) internal structure of constituent objects, membranes and pathLinks marking various subsystems are copied as well]. The operation on chains is shown in figure 5 The description of the action s_X of the splitFork-enzyme to the radical of object X is as follows.

1. A copy X' of X is made.

2. The links incident on X other than loops are distributed among X and its copy as follows:

   - bidirectional links with target X get X' as their target
   - unidirectional links with target X retain X as their target
   - bidirectional links with source X retain X as their source
   - unidirectional links with source X get X' as their source

   The loops $X \mapsto X$ remain in place and get a copy $X' \mapsto X'$.

3. The adjoints of formerly unidirectional links are promoted to the status of links.

One may imagine that the copy X' is connected by a pair of adjoint bidirectional identity links to X to begin with. Then the management of the links can be interpreted as instances of motion - multiplication with identity links and creation of fundamental adjoints. In the end, the identity links between X and X' are killed. This decomposition of the splitfork-enzyme into an action of micro-enzymes is illustrated in figure 6.

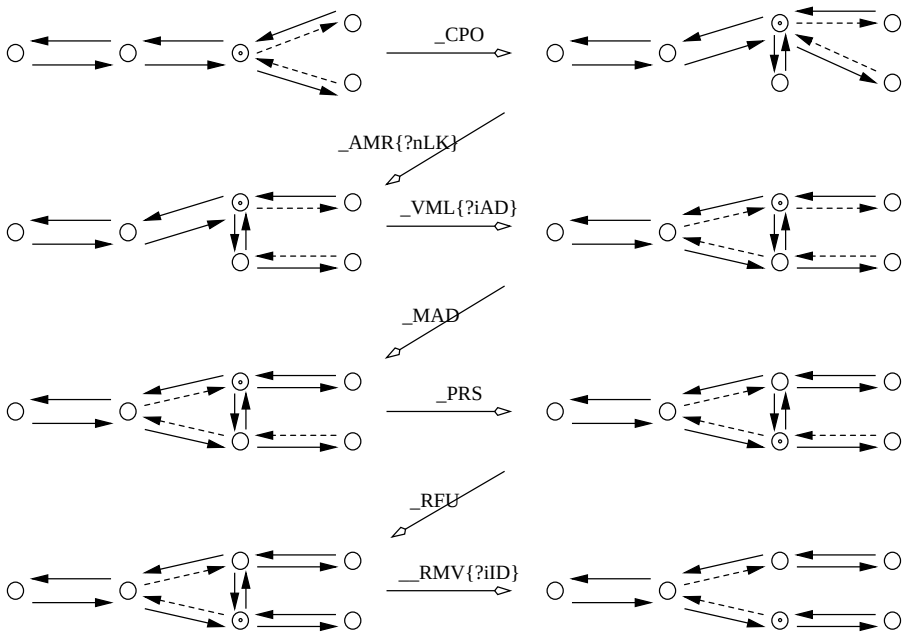


Figure 6: One step within a sweep of the SplitFork enzyme: `sFRK = _CPO _AMR{?nLK} _VML{?iAD} _MAD _PRS _RFU _RMV{?iID} _Sag{?nAD}`

Define a fork at X as a pair of links, both unidirectional, one with target X and the other with source X.

Theorem 1 (Universal copy constructor) *Let* $\mathbf{S}_0$ *be obtained from a finite connected system* $\mathbf{S}$ *whose links are all bidirectional by action of* s_{X_0} *at some* $X_0 \in \mathbf{S}$. *For* $t > 0$, *let* $\mathbf{S}_t$ *be obtained from* $\mathbf{S}_{t-1}$ *by action of* s_X *for all objects* X *such that there is a fork at* X.

$\mathbf{S}_t$ *is well defined for* $t \geq 0$. *For sufficiently large* t, *it is independent of* t *and consists of two disconnected copies of* $\mathbf{S}$.

"Once replication has started, it continues until the entire SYSTEM has been duplicated". Upon substituting "genome" for "SYSTEM" this becomes a quote from a genetics text book [17].

This theorem was first demonstrated in [20].

The fact that s_X is mathematically well defined is somewhat subtle. It rests on a theorem, proven in [21], that a SYSTEM can be specified up to isomorphism by enumerating its arrows, which of them can be composed

and which arrow is the result. One must also indicate which arrows are links and what are their adjoints. But nothing need be said about objects - they can be reconstructed. Using this, the above theorem can be proven, with the understanding that the phrase "two copies of **S**" means "two SYSTEMs, both isomorphic to **S**". The isomorphism class of **S** does not retain the information about the internal structure of non-atomic objects (*black boxes*). But this information can be retained by the copies. To retain it, one uses the universal copy constructor to copy objects of **S** which are themselves SYSTEMs, to copy their non-atomic constituents, and so on. This does not continue *ad infinitum* by the axiom of non-self-inclusion.

The fact that the dynamics is well defined rests on the fact that $s_X s_Y = s_Y s_X$ for the kind of SYSTEMs that occur as $\mathbf{S}_t$.

The action of the splitFork-enzyme is quite robust against errors due to computer failures which mimic local mutations. But there is one exception which leads to catastrophic results of the type of Down's syndrome - a third copy is made of part of the SYSTEM. This happens when a fundamental adjoint gets lost (or added) "at the wrong moment", cf. [20]. [6]

There is a more flexible (but not quite as axiomatic) version of the splitFork-enzyme which can be used to restrict copying to subsystems bounded by membranes. It employs membranes in place of the prototypical membrane made by unidirectional links.

8.2 Digestion

In animals, building blocks of newly made structures must first be prepared by degrading ingested organic material by metabolic processes.

Here we describe an enzyme whose continues action d_X at an object X of a connected SYSTEM degrades its internal structure in the manner shown in figure 8.2.

d_X consists of consecutive steps.

1. (Death) The far side of all triangles of 3 links with tip X is removed, together with their adjoints.

2. (Motion) If b is a link from $Y \neq X$ to X and b' is a link from Y' to Y then $b \circ b'$ becomes a link and b' ceases to exist as a link.

3. Fundamental loops $X \mapsto X$ are removed.

[6] In man, Down's syndrome is caused by the presence of three copies of chromosome no. 21 instead of the usual two

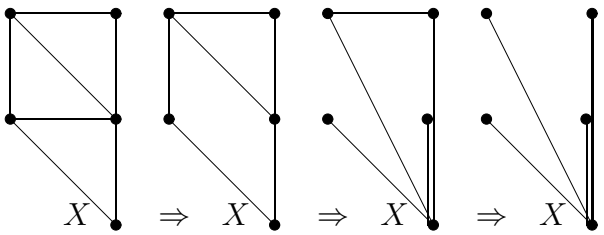


Figure 7: Digestion enzyme attacks at X

Actually the 3rd step can be dispensed with when step 2 operates also for $Y = X$.

8.3 Reaction Diffusion

Restrict attention to those values of objects which can be added and subtracted.

Consider a radical $\mathtt{r}$ with object X and its valences $\mathtt{v}$ with source objects Y_v. There is an enzyme which does the following.

For every valence $\mathtt{v}$ with source $Y_{\mathtt{v}}$ (or those which fulfill the condition set by a predicate) it takes a fraction $\alpha \in \mathbf{R}$ of the original value ξ of X, subtracts it from ξ, parallel transports it to $Y_{\mathtt{v}}$ along $\mathtt{v}$ and adds the result to the value $\eta_{\mathtt{v}}$ of $Y_{\mathtt{v}}$. Conversely, it takes a fraction α of the original value $\eta_{\mathtt{v}}$, subtracts it from $\eta_{\mathtt{v}}$, parallel transports it to X and adds the result to ξ. In the most important examples, the parallel transport is trivial (i.e. ξ, η are invariants). This models diffusion.

One may add loops $X \mapsto X$ which effect nonlinear maps of the values of X as "parallel transports". This models reactions, e.g. chemical reactions.

8.4 Shock fronts

The splitFork dynamics is an example of the mechanism of propagating shock fronts. It is an abstraction of the most essential feature of shock fronts in nature. [7] In the splitFork dynamics, there is at any time a bipartite boundary between the part of the SYSTEM which has not yet been copied, and the two

[7] Shock fronts occur in nature as a consequence of nonlinearity in excitable media [7] when the passage of an excitatory event is followed by a dead time. It has the consequence

copies of the rest. The two boundaries with the two copies have opposite orientation. They are made of the unidirectional fork prongs. The excitation ($sFRK$-action) is restricted to radicals which have forks, and it can only pass through bidirectional links into the part of the system which has not been copied yet.

Similarly, a single boundary made of unidirectional links may be made to propagate unidirectionally. There is an enzyme $aFRK$ to do that. It generates sweeps through the system. Making an enzyme chain with head $aFRK$ and some tail will make the tail act on every radical `r` of the SYSTEM. This may be used to induce relaxation sweeps, for instance.

8.5 Cellular growth

Cell replication can be achieved by realization of the following idea: A cell A gets triggered to replicate, for instance by a certain value of a gradient to a neighboring cell B. A copy C of A is made and placed between A and B in order to smoothen the gradient. Therefore the fundamental links between A and B are replaced by new fundamental links between B and C. Furthermore C obtains new fundamental links to all neighbors, A and B are both connected to (see figure 8).

8.6 Assembly of data

Consider a finite SYSTEM with a distinguished radical `root`. We describe another shock wave mechanism, call it assemblyFork, which can be used to assemble data from the system at `root`.

First one sends out a shock wave which builds a spanning tree - details are given below.

Take any data which are of type *AlgebraicValue*. They may be attached to objects or links. The assemblyFork-dynamics computes the sum of these algebraic values, parallel transported along the branches of the tree, and deposits it at `root`.

For instance, the data maybe sets of strings [8], as occur in encodings of

that the excitation can only propagate in the outward direction because a renewed excitation is prohibited during the dead time. Forest fires are a well known and much studied example [9]. Natural neurons also have a dead time after firing.

[8] sets become lists by lexicographic ordering

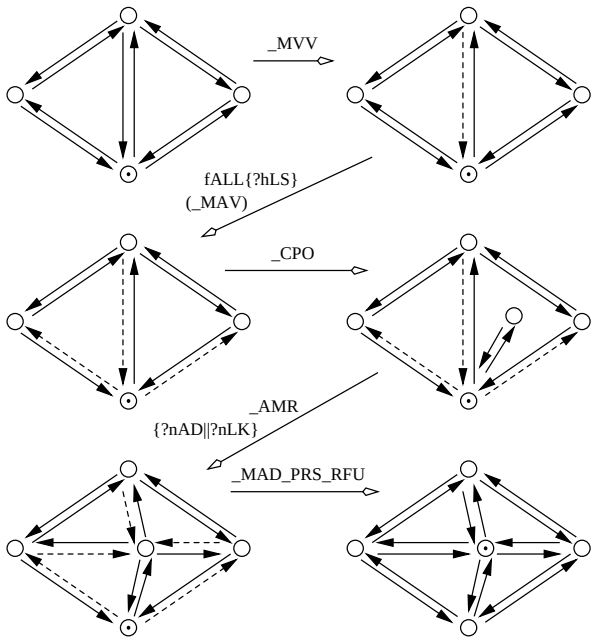


Figure 8: Cell replication in a system theoretic diagram: `cell = _MVV fALL {?hLS}(_MAV) _CPO _AMR {?nAD||?nLK} _MAD _PRS _RFU`

data in **XML**, where $\circ$ is concatenation of strings, and $\oplus$ is disjoint union of sets. No loss of information is involved in taking the sum (disjoint union) in this case.

Or the data may be the (real) elements of a *max-plus*-algebra, where addition amounts to taking the maximum, and multiplication is addition of real numbers. The maximum of all the data is computed in this case.

If the data are invariants (cf. earlier) then, by definition, it would make no difference if parallel transport were along any path to `root` other than the path along the tree. [9]

When a shock proceeds up along any of the branches of the tree, a leaf will eventually be reached where no prolongation of the branch is possible without creating a loop. Then the shock gets reflected and propagates down the tree, taking data from the leaf along via parallel transport (cf. section 5.2). At any node of the tree, the shock has to wait until the reflected shocks

[9]More generally, if all links are unitary, i.e. if $b^* = b^{-1}$, the results differ at most by a gauge transformation, cf. [21].

from all the branches are in. Then the data which they transport are all added up and the result is transported further down the tree.

The *max-plus* version of this mechanism is important when one does optimization by relaxation. One needs to know the maximum of a measure of the local error in order to know when to stop.

Let us finally describe how the spanning tree is made and marked. In principle, the order of the valences in the valence list of a radical is irrelevant. But we may either make an exception for the purpose of the assemblyFork-mechanism, distinguishing the first valence as a "principal port". Or we may use paths to mark the branches of the tree. Suppose we do the first; generalization will be obvious.

Given a distinguished radical *root*, the SYSTEM decomposes into shells labeled by path-distance [10] $n \geq 0$ from `root`, and links between adjacent shells. The shells are like the shells of an onion. The n-th shell is a subsystem which contains all objects with path distance n from *root*, and the links between them.

A shock emanating from `root` propagates from shell to shell in order of increasing n.

When starting at shell 0, the tree has no link.

When the shock reaches shell n, there is a membrane = boundary between shell n and $n+1$, and the tree n will have been constructed up to shell n. Now one selects in some arbitrary way links from the boundary such that each object X of the $n+1$st shell is source of exactly one such link. This can be done by local operations. The adjoint of the unique link with source X is made principal port of X. In this way the tree has been grown to shell $n+1$. Now one continues. In a finite SYSTEM, there are finitely many shells. So the process comes to an end.

9 Outlook

There exists also an (exterior) differential calculus and geometry on SYSTEMs [8, 21]. It is intermediate between standard exterior differential calculus and geometry on manifolds and non-commutative differential calculus and geometry [5]. It satisfies $d^2 = 0$ and the Leibniz rule, the algebra of functions is commutative, and derivatives of functions are ordinary finite difference

[10] path distance was defined in subsection 5.1

derivatives. It has not been implemented in software yet, but we plan to do so in the future.

Another line of current investigation is multiscale analysis [13]. By definition, genuinely complex systems show "emergent" properties that are not shared by their subsystems with few objects. The basic idea of multiscale analysis, including multigrid methods[13] and the renormalization group [29], is that although a genuinely complex system can not be understood as a whole by studying reasonably small subsystems in isolation, a complexity reduction is often possible by doing so. To this end one introduces new objects which represent subsystems, but one retains only that part of the information on their internal structure (including functionality, i.e. enzymes) that is relevant for the cooperation that causes emergent - i.e. nonlocal - phenomena. A related plan is known in information science under the name of *integration* [10]. Some theoretical considerations concerning this were presented in [21] and in [28]. Relaxation sweeps can be implemented by enzymatic computation, and it is straightforward to extend them to SYSTEMs which have a multilevel structure like multigrids. The challenge is to extend the domain of applicability of existing multigrid and renormalization group technology, including in particular disordered systems.

Finally it is a challenge to invent an "enzymatic game of life", and to implement within the present frame as much of biochemical and larger scale processes that constitute life. This would be in the spirit of work on Artificial life [16].

9.1 Towards an infome project

A much more challenging and long range plan would be an *infome* project which amounts to mapping into software what is known about structure and function in the living cell in such a way that processes that constitute life can be simulated *in silico*. After the genome project and the envisaged proteome project [32] which is supposed to classify all proteins that a human body makes in its lifetime and their function, this is a logical next step.

Work on immunology along similar lines is in progress [23]; the establishment of a data base has been the proposed. For a discussion of the cognition problems that are of interest in immunology see [24].

10 Acknowledgement

We thank the Deutsche Forschungsgemeinschaft for financial support, and the German Israel foundation for a travel grant. We would like to express our thanks to Achi Brandt, Irun Cohen, Martin Meier-Schellersheim, Dirk Rathje, Sorin Solomon, and York Xylander for numerous enlightening discussions. We also thank Daniel Lübbert, Bleicke Holm, Marcus Speh and York Xylander for their collaboration in earlier stages of this project.

References

[1] L.M.Adleman, *Molecular Computation of Solutions to Combinatorial Problems*, Science **266** 1021 (1994)

[2] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, J. Watson. Molecular Biology of The Cell. Garland Publishing, Inc. 1989.

[3] L. von Bertalanffy, *Les problèmes de la vie*, Paris:Gallimard 1956
- , *General systems theory*, George Braziller, New York 1968

[4] Hans Braker, *Algorithms and Applications in Timed Discrete Event Systems*, Dissertation Defft 1993
C. Pöppe, *Fahrplanalgebra*, Spektrum der Wissenschaft, Digest Wissenschaftliches Rechnen 1999

[5] A. Connes, *Noncommutative Differential Geometry*, San Diego: Academic Press 1994

[6] M. Creutz, *Quarks, Gluons and Lattices*, Cambridge (UK): Cambridge University Press 1983

[7] M.J. Cross, P.C. Hohenberg, Rev. Mod Phys. **65** (1993) 851
H.L. Swinney, V.I.Krinski (eds), *Waves and patterns in chemical and biologicl media*, Physica **D49** (1991) 224
J.P. Keenes, J.J. Tyson, Physica **D32** (1988) 327,
J.L. Martiel, A. Goldbeter, Biophysics **J52** (1987) 807
J.Parisi, S.C. Müller, W.Zimmermann (eds), *Nonlinear physics of complex systems - current status and future trends*, Springer Berlin 1988
- *A perspective look at nonlinear media - from physics to biology and social sciences*, Springer, Berlin 1988

[8] A. Dimakis, F. Müller-Hoissen, *Differential calculus and gauge theory on finite sets* J. Phys. **A27** (1994) 3159
-, *Discrete differential calculus, graphs, topologies and gauge theories*, J. Math. Phys **35** (1994) 6703
A. Dimakis, F. Müller-Hoissen, F. Vandersypen, *Discrete differential manifolds and dynamics on networks*, J. Math. Phys **36** (1995) 3771
D. Weingarten, *Geometric formulation of electrodynamics and general relativity in discrete space-time*, J. Math. Phys. **18** (1977) 165

[9] B. Drossel, *Strukturbildung in offenen Systemen am Beispiel eines Waldbrandmodells*, Reihe Physik Bd *21*, Harri Deutsch, Frankfurt

[10] H. Ehrig and F. Orejas, *Integration and Classification of Data Types and Process Specification Techniques*, Bulletin EATCS: Formal Specification Column 65 (1998)

[11] R.P. Feynman, in *Miniaturization*, D.H. Gilbert, Ed. Reinhold, New York 1961) pp 282-296
- , *The Feynman Lectures On Computation*, A.Hey, R. Allen (ed.), Penguin Books, 1999

[12] E. Gamma, R. Helm, R. Jahnson, J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] W. Hackbusch, *Multigrid Methods and Applications*, New York:Springer 1985
U. Trottenberg, C. OOsterlee, A. Schüller, *Multigrid*, New York:Academic Press 1999

[14] H. Haken, *Synergetics*, Heidelberg: Springer 1978
M.M. Waldrop, *Complexity: The emerging Science at the edge of Order and Chaos*, New York: Simon and Shuster 1992

[15] F. Jacob, *La logique du vivant*, Paris:Gallimard (1970)

[16] C.G. Langton, *Artificial Life*, SFI Studies in Sciences of complexity, Redwood City: Addison Wesley (1989)

[17] B. Lewin, *Genes VII*, Oxford (UK):Oxford University Press 200, p. 349

[18] R. Light, *Presenting XML*. Sams.net Publishing, 1997.

[19] S. Mac Lane, *Categories for the working mathematician*, Springer Verlag, New York 1971

[20] G. Mack, *Gauge theory of things alive: Universal dynamics as a tool in parallel computing*, Progress Theor. Phys. (Kyoto) Suppl. **122** (1996) 201-212

[21] G. Mack, *Universal Dynamics, a Unified Theory of Complex Systems. Emergence, Life and Death*, DESY 00-146, hep-th/0011074, to appear in Commun. Math. Phys.

[22] G. Mack, *Pushing Einsteins principles to the extreme*, in: G. 't Hooft et al, *Quantum Fields and Quantum Space Time*, Plenum Press, NATO-ASI series B:Physics vol. 364, New York 1997, gr-qc/9704034

[23] M. Meier-Schellersheim, G. Mack, *SIMMUNE, a Tool for Simulating and Analyzing Immune System Behaviour*, submitted to Bull. Math. Biology, cs. MA / 9903017.

[24] A. S. Perelson and G. Weissbuch, Immunology for Physicists, Rev. Mod. Phys. 69, No.4, 1997.
I. Cohen *Tending Adam's garden: Evolving the Cognitive Immune Self*, Academic Press, New York 2000

[25] C.A. Petri, *Kommunikation mit Automaten*, PhD-thesis Bonn, Schriften des Instituts für Instrumentelle Mathematik (1962)

[26] G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation. vol 1: Foundations*, Singapore:World Scientific 1997

[27] M. Schrattenholzer, *Logik lokal*, Diplomarbeit Hamburg 1999

[28] T. Tomkos, *Formal specification and modelling of Complex Systems. Towards a physics of information via networks*, phd thesis Hamburg 1999.

[29] K. Wilson, *Renormalization group and strong interactions* Phys. Rev **D3** (1971) 1818
J. Kogut, K. Wilson, *The renormalization group and the ϵ expansion*, Phys. Reports **12C** (1974) 75

[30] S. Wolfram (ed) *Theory and Applications of Celluar Automata*, Singapore: World Scientific 1986

[31] Jan Würthner. *Enzymatic Simulation of Complex Processes*, ph.d. thesis, Hamburg, Feb. 2000, accessible via http://lienhard.desy.de . The most current version SyCL.xx of the software will be made accessible on the same site.

[32] A. Pandey and M. Mann, *Proteomics to study genes and genomes*, Nature **405** (2000) 837-846
Proteomics: A Trend guide, Suppl. to Trends in Microbiology, July 2000