Transformation Semigroups as Constructive Dynamical Spaces

Attila Egri-Nagy[1], Paolo Dini[2], Chrystopher L. Nehaniv[1], and Maria J. Schilstra[1]

[1] Royal Society Wolfson BioComputation Research Lab
Centre for Computer Science and Informatics Research
University of Hertfordshire
Hatfield, Hertfordshire, United Kingdom
{a.egri-nagy,c.l.nehaniv,m.j.l.schilstra}@herts.ac.uk
[2] Department of Media and Communications
London School of Economics and Political Science
London, United Kingdom
p.dini@lse.ac.uk

Abstract. The informal notion of constructive dynamical space, inspired by biochemical systems, gives the perspective from which a transformation semigroup can be considered as a programming language. This perspective complements a longer-term mathematical investigation into different understandings of the nature of computation that we see as fundamentally important for the realization of a formal framework for interaction computing based on algebraic concepts and inspired by cell metabolism. The interaction computing perspective generalizes further the individual transformation semigroup or automaton as a constructive dynamical space driven by programming language constructs, to a constructive dynamical 'meta-space' of interacting sequential machines that can be combined to realize various types of interaction structures. This view is motivated by the desire to map the self-organizing abilities of biological systems to abstract computational systems by importing the algebraic properties of cellular processes into computer science formalisms. After explaining how semigroups can be seen as constructive dynamical spaces we show how John Rhodes's formalism can be used to define an Interaction Machine and provide a conceptual discussion of its possible architecture based on Rhodes's analysis of cell metabolism. We close the paper with preliminary results from the holonomy decomposition of the semigroups associated with two automata derived from the same p53-mdm2 regulatory pathway being investigated in other papers at this same conference, at two different levels of discretization.

1 Introduction

This expository paper has several goals and consists of three main parts. The first part concentrates on computer science and aims to show that transformation semigroups can provide a theoretical background for programming languages. The second part introduces the concept of interaction computing and discusses a possible architecture

F.A. Basile Colugnati et al. (Eds.): OPAALS 2010, LNICST 67, pp. 245–265, 2010.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2010

for the Interaction Machine based on examples from cell biology. The third part streamlines the theory further and applies some of the algebraic results to the analysis of the p53-mdm2 regulatory pathway [24], as part of our on-going effort to understand and formalize the computation performed by the cell.

This paper is part of a research framework that is documented in the following four companion papers at this same conference:

- A Research Framework for Interaction Computing [7]
- Numerical and Experimental Analysis of the p53-mdm2 Regulatory Pathway [24]
- Lie Group Analysis of a p53-mdm2 ODE Model [16]
- Transformation Semigroups as Constructive Dynamical Spaces (this paper)
- Towards Autopoietic Computing [4]

1.1 The Programming Language Perspective

In the first part of the paper (Section 2) we would like to argue that:

1. Finite state automata, and thus transformation semigroups, have much wider applicability than it is thought traditionally.
2. A transformation semigroup is analogous to a programming language; therefore, whenever a piece of software can model some phenomenon, so does a semigroup.
3. There are different ways to achieve parallel computation, and they all fit naturally into the algebraic framework.
4. The presence of symmetry groups in a computational structure indicates a special kind of reversibility, which is not to be mistaken for the general idea of reversibility.

A *constructive dynamical space*[1] determines a set of possible processes (computations) and provides basic building blocks for these possible dynamics, equipped with ways of putting the pieces together. With these tools one can explore the space of possibilities; or, going in the other direction, given one particular dynamical system it is also possible to identify its components and their network of relations. A prime example is a programming language (complete with its runtime system): we build algorithmic (thus dynamical) structures using the language primitives in order to model or realize the dynamics that we are interested in. In this paper our main purpose is to show that finite state automata and transformation semigroups are other examples of constructive dynamical spaces, although they are not usually considered as such. The possible gain is that we can bring the algebraic results, mainly the hierarchical decomposition theories, into domains of applications outside mathematics. For example, currently programming languages do not include tools for automatic decomposition and reconstruction of the problem domain.

The term is used in artificial life research for artificial chemistries (e.g. [2]), and this does agree with our definition.

1.2 Interaction Computing

The second part of the paper (Section 3) extends the perspective on an individual automaton to two or more interacting automata, and explores the implications of such

---

[1] The term is used in artificial life research for artificial chemistries (e.g. [2]), and this does agree with our definition.

a generalization. Figure 1 shows the broader theoretical context within which this paper is situated. The study of semigroups as programming languages fits within the activity labelled as “Algebraic properties of automata structure” in the figure. The concept of interaction computing can be seen as an area of application that aims to combine *interacting* constructive dynamical spaces to create an environment analogous to the cell’s cytoplasm. In particular, Section 3 considers the following points:

1. Interaction computing builds on ideas that have been around since Turing’s 1936 paper [23] and that require a shift in how we think about computing.
2. The Interaction Machine can be given a formal foundation and a high-level architecture based on Rhodes’s work [22].
3. The groups found in the hierarchical decomposition of the semigroups associated with the automata derived from cellular pathways suggest a form of parallel computation that relies on cyclic phenomena and on interdependent algorithms (i.e. symbiotic cohesion, and generally opposite to loose coupling).

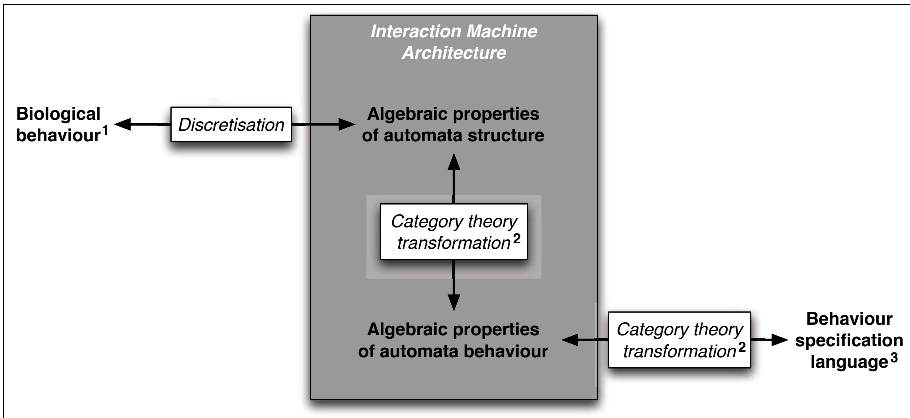

Fig. 1. Theoretical areas of relevance to interaction computing whose exploration is discussed in this and other papers: 1 = [24]; 2 = [8, 6, 21, 7]; 3 = future projects/papers; no superscript = this paper.

The term ‘interaction computing’ captures the essence of a particular form of biologically-inspired computing based on metabolic rather than evolutionary processes. Specifically, the concept of interaction computing is based on the observation that the computation performed by biological systems always involves at least two entities, each of which is performing a different, and often independent, algorithm which can only be advanced to its next state by interaction itself. The aim of the interaction computing approach is to reproduce in software the self-organizing properties of cellular processes.

For this concept to make sense in a biological context one needs to choose an appropriate level of abstraction. In particular, our perspective views the stochastic nature of cell biochemistry mainly as a mechanism of dimensional reduction that does

not necessarily need to be emulated in any detail. For example, a gene expresses hundreds of mRNA molecules which, in turn, engage hundreds of ribosomes for no other reason than to maximize the probability that a particular, single genetic instruction will be carried out as a single step in a metabolic process, such as the synthesis of a particular enzyme. As a consequence of this dimensional reduction (hundreds to 1), a higher level of abstraction than that at which stochastic molecular processes operate is justified in the modelling approach – in particular, a formalization that retains, and builds on, the discrete properties of cell biology.

However, even the resulting lower-dimensional system can't plausibly be imagined to perform the complexity of a cell's functions driven simply by a uniform distribution of interaction probability between its (now fewer) components. Additional structure and constraints must be at play, for example as provided by molecular selectivity. Whereas the evolutionary processes that have led to molecular selectivity and the underlying physical processes that 'fold and hold' the relevant proteins and molecules together are fascinating phenomena that can be recognized as the ultimate and the material *causes* of order construction in biology, respectively, this does not entail that it is necessary to reproduce these mechanisms to arrive at self-organizing formal systems. We think it is sufficient to recognize the *effects* of these phenomena as embodying the essence of the cell's discrete behaviour as a kind of computation, whose ordered properties can be formalized through algebra. This is the motivation for our work in the development of an algebraic theory of interaction computing. A formal foundation for this theory has been provided by Rhodes's work [22].

1.3 Analysis of the p53-mdm2 Regulatory Pathway

The above ideas for how constructive dynamical spaces and interaction computing could be realized to improve the flexibility and self-* properties of software have only begun to be addressed. As explained in [7] and shown in Figure 1, using category theory we need to develop a mapping to relate the specification of behaviour to the algebraic structure(s) needed to realize it. But before this mapping can be developed we first need to understand in greater depth the algebraic structures that underpin the observed self-organizing properties of biological behaviour. Therefore, the third part of this paper (Section 4) looks in greater depth at the algebraic structure of the semigroup associated with the automata derived from a particular system, the p53-mdm2 regulatory pathway [24], for two levels of resolution in its discretization.

2 Theoretical Framework

2.1 Finite Automata as Computational Spaces

A finite state automaton and its algebraic counterpart, a transformation semigroup, is a computational object, i.e. something that computes. This is not at all surprising, especially from the extreme computationalist's viewpoint ("everything, that changes, computes") and from a classical source in cybernetics [1], but the statement still needs further explanation: *In what sense does an abstract algebraic structure entail computation?*

Formal definition. A finite automaton can be defined as $\mathcal{A} = (A, Q, \lambda)$ where A is the set of *input symbols*, the *input alphabet*; Q is the set of *states*, the *state space* ; and λ is the *state transition function* $\lambda : Q \times A \to Q$. Since we are talking about a finite state automaton, all the objects involved are finite.

Static versus dynamical view. Traditionally, finite automata appear in computability and formal language theories [14]. For problems in those fields we need to extend the definition by giving initial and accepting states. Moreover, an output alphabet can be given together with an output function mapping a state and an input symbol pair to an output symbol. But here our main interest is not using the automaton for a particular purpose, e.g. recognizing certain languages, but to study the automaton itself as a *discrete dynamical system*. However, by looking at the minimalistic definition of finite automata, it is obvious that an automaton is not *dynamical*, i.e. nothing moves in it. Its dynamics, rather, is *potential*, the automaton creates room for possible 'movements'. Applying input symbols to states may move them to other states according to λ, producing a trajectory, but there is no predescribed dynamics of the automaton. The automaton itself rather is merely the setting for any one of an entire space of possible behaviours; therefore, we need to supply the machine with instructions, i.e. we have to provide a starting state and a sequence of input symbols in order to observe change, computation. We can combine the atomic transitions by concatenating input symbols that are interpreted by λ sequentially. This way the series of operations denoted by a sequence of symbols becomes an algorithm, and λ gives meaning to the symbols, thus λ acts as an interpreter. A computational problem can be formalized generally as getting to state r from state q, or producing output r from input q (the states q and r can be arbitrarily complex and r may not be known when starting the algorithm). An algorithmic solution for this problem is

$$q \cdot a_1 \cdots a_n = r \tag{1}$$

where $a_1,\ldots,a_n$ is the sequence of steps of the algorithm.[2] At this point we encounter some possible terminological confusion. So far the input of the automaton was the input symbol, but in the above example we used the state as the input. Looking at λ, the 'machinery' of the automaton reveals that indeed it has two kinds of inputs as a state transition function: the state and the input symbol. This is very important since this view departs from the traditional interpretation. So we can talk about states as

[2] To exploit automata computational capacity further, one might also generalize (1) so that in addition to a sequence of atomic operations, $a_1 \ldots a_n$, one allows a sequence of atomic operations and variables that each evaluate to some basic operation based an additional component of input (i.e. other inputs than state q). That is, one has

$$q \cdot t_1 \cdots t_n = r \tag{2}$$

where each t_i is either an input symbol $a \in A$, or a variable v, ranging over A, whose value is determined by some 'environmental' input. The generalization allows one to make use of so-called *functional completeness* properties of simple non-abelian groups that allow them to compute any finitary function (similar to the two-element Boolean algebra). See [18, 15]. For simplicity, we do not pursue this promising variant further in this particular paper.

inputs, like data fed into the algorithm. This looks quite natural if we think in terms of a universal computer: we have to provide both the algorithm and the input data for the algorithm to work on.

This concludes the introduction of the metaphor

$$\text{finite state automaton} \equiv \text{programming language.}$$

Before continuing, let's summarize: Q, the set of states of the automaton A, is the problem domain in which we would like to solve problems. These problems include going from a start state to a target, a 'desired' state, or a solution. We can also consider states as bits of data and do calculations on them, so the 'number crunching' aspect of computation also appears here. A, the set of input symbols, contains the language primitives, i.e. the basic commands that can be combined into longer sequences to form programs. The transition function λ is then the machinery: the compiler, interpreter, runtime system with libraries, processor, etc. In short,

states Q	problem domain
input symbols A	basic commands
input strings A^+	algorithms
$w : Q \to Q,\ w \in A^+$	computational function
transition function λ	runtime system

We need to differentiate between the automaton and the mathematical function (machine) that the automaton implements (realizes), since the same computational function (same state mapping) can be expressed by different algorithms (sequences). In other words, following Rhodes's terminology, the machine is the behaviour while the automaton or circuit is what implements it.

2.2 Problems, Critique

Problem: "Finite state automata are too constrained compared to general-purpose programming languages". At first sight an automaton as a programming language seems to be quite constrained. We show that the only constraint is finiteness, and this poses no practical problems. Usually programming languages are all considered to be Turing-complete, thus being in a completely different computational class than the one formed by finite automata. This is only true if we assume unbounded resources for the machine on which the programming language is implemented. But if we consider a physically existing (hence finite) computer, then we have a finite state automaton. Its state set may be enormous but still finite. For example, one state of a computer with 1 gigabyte memory can be described by 8796093022208 bits, therefore it has $2^{8796093022208}$ distinct states, so the number of possible state transformations on this computer is

$$(2^{8796093022208})^{2^{8796093022208}},$$

which is quite a big number. Of course this large collection of possible computational functions contains the workings of all possible operating systems, with all possible applications as substructures. The situation is reminiscent of the Library of Babel [3].

One of the great capabilities of the human mind is that it can look at things from different perspectives, ignoring the view in which they normally appear. This is really needed here as we usually meet finite automata only of moderate size and usually in textbooks (probably the largest ones in natural language processing), but still not in the magnitude mentioned above. Thus, *finiteness is the only constraint*, and that also applies to programming languages implemented on computers.

"Finite state automata-based programming is a specialized technique mainly for implementing lexical and syntactic analysers". Our approach is not about one particular technique that can be applied in a programming situation (like using state transition tables in an algorithm), but a more general conceptual framework to enable us to see the applications of automata in more general settings, in dynamical systems.

2.3 A More Comprehensive View: Transformation Semigroups

A^+ forms a semigroup under the operation of concatenation, and its elements can be interpreted, using λ recursively, as different mappings of the state set to itself. The finite set of these mappings is called the semigroup $S(\mathcal{A})$ corresponding to the automaton $\mathcal{A}$ and these mappings can be combined by function composition (i.e. following one by another). In (1), a computation was just one run of the algorithm on a particular input x (in (1) this input is the state q). The same algorithm can be applied to other inputs (states) as well. Therefore, each a $\in A$ is an (atomic) algorithm which yields a (single) mapping $Q \rightarrow Q$; the latter is a (generating) element of the corresponding transformation semigroup $S(\mathcal{A})$. So semigroup elements realize algorithms defined for all possible inputs; therefore, a semigroup is a "constructive space of total algorithms", which is basically a programming language.

Built-in primitives ≡ generator sets. Programming languages are equipped with different sets of built-in basic tools; thus the same algorithm requires longer code in a low-level language than in a high-level one. Analogously, a transformation semigroup with a smaller set of generators will have longer words to express the same semigroup element. Different semigroups can have common elements, just like different programming languages can share the same tools and techniques. The extreme case is the automaton where we have a symbol for each semigroup element, corresponding to a language in which there is a language primitive for each expressible algorithm.

Self-modifying algorithms. This powerful idea is incorporated in some languages (e.g. LISP), but it is not a mainstream property. In the algebraic settings it is natural as semigroup elements can be multiplied by each other, thus algorithms can act on each other (algorithms as inputs and outputs of other algorithms), thereby modifying themselves. This is just a consequence of the generalization of Cayley's theorem to semigroups.

2.4 Going Parallel: The Role of Permutation Groups

Once we start talking about transformation semigroups, the special class of permutation groups naturally comes into the picture. As a consequence, we need to give an account of symmetries in a computational context.[3]

Reversibility. A permutation of the state set (or a subset of it) as a totally defined algorithm has the peculiar property that, when applied to the whole set in parallel, it gives back the whole set: no two states/inputs are collapsed into the same output. This is like maintaining the set of possible future states, not losing any possibility. This prompts us to believe that algebraic groups of algorithms coincide with the notion of reversibility. This is certainly true if we consider the parallel movement of all states, but it becomes subtle if we study individual inputs: permutations are not the only way to achieve reversibility. The following examples will show the different kinds of reversibility.

Example: Reversibility by resets. Let's consider state 1 of the state set $\{1, 2\}$ and two transformations $t_1 = \binom{1\ 2}{1\ 1}$, $t_2 = \binom{1\ 2}{2\ 2}$. t_2 takes 1 to 2, and we can reverse this movement since we can go back to 1 by t_1 . t_1 , t_2 are called constant maps, or resets.

Example: Cyclic reversibility. Similarly, by applying the permutation (1, 2) (or p = $\binom{1\ 2}{2\ 1}$) we can go from 1 to 2: $1 \cdot p = 2$, but we can get back to 1 by applying p again: $1 \cdot pp = 1$. So we achieve reversibility by 'going forward', by a kind of "cyclic reversibility". This can be achieved only by permutations.

Cycles. Another interpretation of the symmetries that algebraic groups symptomize is the presence of periodic cycles of reactions in the metabolic systems from which the automata are derived, the invariant feature being the existence of a stable cycle that is preserved by the elements of the group. This too is compatible with the parallel interpretation of the algorithm arising from a group. In fact, conceptually, a group element can be conceived to act on all the states at once, and this can be implemented as a set of parallel automata (as many as there are states), all transitioning simultaneously and in parallel. Such a model is entirely consistent with periodic pathways such as the Krebs cycle, whose equilibrium operation is characterized by all the reactions taking place in parallel, at the same average rate. In such a case a group element can be considered as realizing a parallel algorithm on several inputs. This can be viewed on different levels:

1. acting on the whole Q
2. acting on the subsets of Q
3. acting on multisets of elements of Q

More generally, there are other natural possibilities too (for any positive integer n): n-tuples of elements of Q, n-tuples of subsets of Q, or n-tuples multisets of elements of

[3] This is because of two facts: (1) a symmetry of an object is a transformation that leaves some aspect of that object invariant; and, (2) the set of invertible transforma- tions that leave some aspect of a mathematical object invariant form a group with the operation of functional composition.

Q, not to mention coproduct structures, etc. As transformations are the computational functions realized by totally defined algorithms, the definitions of these actions can be derived in a natural way from the action of transitions symbols of the automaton.

2.5 The Gain: Algebraic Results

After we presented the analogy, finally we can show the benefits of thinking about the automata in a different way.

Hierarchical coordinatization. When dealing with huge automata we need some tools for organizing their structure. Just as, when analysing software, we find modules and subroutines that are combined in a hierarchical way, so too we can decompose semigroups with hierarchical coordinates [20].

The disappearance of memory–processor duality. The algebraic theory of machines gives us a level of abstraction at which we can consider both processing and memory units having the same nature, namely they are semigroups. For storing single bits of information the semigroup of the flip-flop automaton can be used and for calculations the permutation groups can be used. In the hierarchical construction of automata this allows us to dynamically change the ratio of computational power and storing capacity on demand, as in real computational problems either the power of the processor or the insufficient amount of memory is the bottleneck. For exploiting this idea in practical computing we would need some physical implementation, e.g. reconfigurable hardware that is capable of dynamically allocating automata structures.

3 The Interaction Machine Concept

The above discussion provides a theoretical backdrop against which we can now propose some novel ideas about computing, in particular interaction computing. The concept of interaction computing was originally inspired by a physics perspective on biological systems ([5, 7] and references therein); in other words, in looking for general principles that appear to govern self-organizing behaviour in cell biology, the role of interactions appeared to be such a fundamental feature that it seemed indispensable to replicate it in computational systems in order to develop an 'architecture of self-organization' in software along analogous principles.

The addition of an algebraic automata theory perspective to the above stream of work has opened the possibility to develop a formalism that can express the behaviour of biological systems seen as dynamical systems in a manner that is consistent with the mathematical foundations of computer science. We now discuss a few examples that are helping us understand how the algebraic properties of automata discussed so far can be interpreted in the context of 'biological computation', and how this mapping between algebra and biology is helping us imagine the architectural requirements of the 'Interaction Machine'.

3.1 Sequential Machines and Their Realizations

In this section we now consider not only computations inside individual transformation semigroups (as in Section 2), but also (1) their behaviours (abstracting

away states) as formalized by *sequential machines*, which is closely related to software specification, and (2) a higher-level constructive dynamical ‘meta- space’, one in which sequential machines are basic entities that may interact and combine in various ways to create new kinds of computational structures; in particular, they extend to *interacting machines*, whose deployment together can create complex interaction structures.

Many years after proving the prime decomposition theorem for semigroups and machines [19], John Rhodes published a book that he had started working on in the 1960s and that has come to be known as the ‘Wild Book’ [22]. In this book he provides a very clear discussion of an alternative to Turing machines, which we believe to be a very promising starting point for a model of interaction computing.

As we know, an algorithm implementable with a Turing machine is equivalent to the evaluation of a mathematical function. As Wegner and co-workers argued in a series of papers over the last 20 years ([13] and references therein), the evaluation of a mathematical function can afford to take place by following an ‘internal clock’, i.e. the Turing machine is isolated from its environment while it evaluates the algorithm. Biological systems, on the other hand, are continually interrupted by external inputs and perturbations. As an example of this class of computations Golding and Wegner used ‘driving home from work’, which they described as a non-Turing-computable problem. Turing himself had foreseen this possibility in his original 1936 paper as the ‘choice machine’ [23], although he did not pursue it further.

Similarly, Rhodes starts from the familiar definition of a sequential machine. The sequential machine accepts an input at each discrete point in time and generates an output once all the inputs have been received:

> Let A be a non-empty set. Then $A^+ = \{(a_1, ..., a_n) : n \geq 1 \text{ and } a_j \in A\}$. A *sequential machine* is by definition a function $f : A^+ \rightarrow B$, where A is the basic input set, B is the basic output set, and $f(a_1, ..., a_n) = b_n$ is the output at time n if a_j is the input at time j for $1 \leq j \leq n$.

This is clearly related to the formal definition of the finite automaton given at the beginning of Section 2, but there is a twist: Rhodes prefers to make a sharp distinction between a *machine*, which he equates to a *mathematical function*, and the *realization of that machine*, which he calls a *circuit* and that is essentially an *automaton*:

Mathematical concept . . .	**. . . and its realization**
Machine or mathematical function	Circuit or automaton
Automata behaviour	Automata structure

Due to the need to maintain the development of a theory of interaction computing on firm mathematical grounds, we follow his approach. More specifically, the separation between a machine and its realization matches well the distinction between the description (formalization) of *behaviour* and the automaton *structure* necessary to achieve it. It is essential for us to maintain this distinction in light of a parallel thread of research [7, 6] which applies categorical morphisms to automata behaviour in order to derive a specification language. To understand better what we might be aiming to specify, let’s develop the idea further.

We are going to use a generalization of the sequential machine, also by Rhodes, which produces an output for each input it receives and not just in correspondence of the most recent input. We are going to call this generalization an interact**ing** machine:

> Let $f: A^+ \rightarrow B$ be a sequential machine. Then an *interacting machine* f^+ : $A^+ \rightarrow B^+$ is defined by $f^+(a_1, ..., a_n) = (f(a_1), f(a_1, a_2), ..., f(a_1, ..., a_n))$.

Thus, an Interact**ion** Machine (IM) can be built by joining two or more interact**ing** machines. Such an IM will still accept inputs from outside itself ("the environment") and will produce outputs for the environment. The realization of either machine is achieved through a finite-state automaton that Rhodes calls a "circuit", C , but that in the literature is more commonly called a Mealy automaton:

> C = (A, B, Q, λ, δ) is an *automaton* with basic input A, basic output B, states Q, next-state function λ, and output function δ iff A and B are finite non-*empty sets, Q is a non-empty set,* $\lambda : Q \times A \rightarrow Q$, *and* $\delta : Q \times A \rightarrow B$.

Having established then that the problem of computation is posed in two parts, a mathematical function and its realization, we continue to rely on Rhodes to define a few more concepts related to the latter, in order to develop a relatively concrete working terminology. The next concept we need is the realization of an algorithm, as follows. Let $C = (A, B, Q, \lambda, \delta)$ be an automaton. Let $q \in Q$. Then $C_q : A^+ \rightarrow B$ is the *state trajectory associated with state q* and it is defined inductively by

$$C_q(a_1) = \delta(q, a_1)$$
$$C_q(a_1, ..., a_n) = C_{\lambda(q,a_1)}(a_2, ..., a_n), \text{ for } n \geq 2.$$

We say that C *realizes* the machine $f : A^+ \rightarrow B$ *iff* $\exists q \in Q : C_q = f$. By a simple extension of the above definitions it is fairly easy to see that the output of an algorithm can be associated with a sequential machine when the output corresponds to the result after the last input, whereas it is associated with an interacting machine when there are as many outputs as there are inputs:

$$C_q(a_1, ..., a_k) = b_k, \qquad \text{for } k = 1, ..., n$$
$$C^+_q(a_1, ..., a_n) = (b_1, ..., b_n).$$

Rhodes then introduces more formalism to define precisely a particular automaton that realizes a function *f* as C (*f*), and goes on to prove that C (*f*) is the unique minimal automaton that realizes *f*. He goes on to say

> The reason why Turing machine programs to realize a computable *f* are not unique and the circuit which realizes the (sequential) machine *f* (namely *C* (*f*)) is unique is not hard to fathom. In the sequential machine model we are given much more information. It is 'on-line' computing; we are told what is to happen at each unit of time. The Turing machine program is 'off-line' computing; it just has to get the correct answer – there is no time restraint, no space restraint, etc. ([22]: 58)

Rather than constructing dynamical behaviour through a sequential algorithm expressed in a programming language, which can be realized by a single automaton or Turing machine, we are talking about constructing dynamical behaviour through the interaction of two or more *finite*-state automata. This is because, if the possibility that Q be infinite is left open as the above definition does, "then the output function could be a badly non-computable function, with all the interesting things taking place in the output map δ, and we are back to recursive function theory" ([22]: 59). Therefore, we note the interesting conclusion that, for a tractable approach, Q must be finite and that the realization of an Interaction Machine must be made up of interacting finite-state automata. Thus, from the mathematical or behavioural perspective, we will build the Interaction Machine by using sequential (or interacting) machines as basic units and by combining them in various ways.

The mathematical and computer science challenge, therefore, is to develop a formalism that is able to capture the non-linear character of the dynamics of an arbitrary number of coupled metabolic systems, so that when this mathematical structure is mapped to our yet-to-be-developed specification language we will be able to specify software *environments* capable of supporting self-organizing behaviour through interaction computing, driven by external stimuli from users or other applications.

In the next subsection we switch the perspective from synthetic, i.e. *constructing* automata and models of computation, to analytical, i.e. *analysing* cellular pathways and interpreting the role and function of the algebraic structures they harbour.

3.2 Permutation Groups from the Biological Point of View

The presence of permutation groups in the hierarchical decomposition of the semigroups associated with automata derived from cellular pathways presents a difficult puzzle. This is because a permutation group can be interpreted in different ways: the fact that groups are defined up to isomorphism means that different 'implementations' could be derived from the same abstract group. In this subsection we explore some of the possibilities. Ultimately, if we are not able to resolve this puzzle theoretically, *ex ante*, it will be resolved *ex post* by building different kinds of interaction machines, based on different interpretations of these groups, and by seeing which kind behaves in the most convincingly 'biological' way.

Based on the analysis of the Krebs metabolic cycle and the p53-mdm2 regulatory pathway, it appears that the permutation groups may be associated with cyclic phenomena in biochemical processes. The challenge is to relate such cyclic or oscillatory behaviour to the computational properties of its mathematical discretization.

It is helpful, first, to clarify the connection between cycles and oscillations by resorting to an idealized system in the form of the simple harmonic oscillator of elementary physics. As shown in Figure 2, for a simple harmonic oscillator periodic cycles (in some parameter space, which could include also Euclidean space) are mathematically indistinguishable from oscillations (in time, at a fixed point in space). It is possible, therefore, that the same kind of algebraic structure, i.e. non-trivial permutation groups, could result from different kinds of cyclic biochemical phenomena. For example, the discrete algebraic analysis discussed in the next section

indicates that biochemical processes such as the Krebs cycle, the concentration levels of whose metabolites can be assumed to remain constant over time, have similar 'algebraic signature' (i.e. permutation groups) to processes such as the p53-mdm2 pathway [24, 16], in which the concentrations of the compounds can oscillate (for some parameter values) as a function of time.[4]

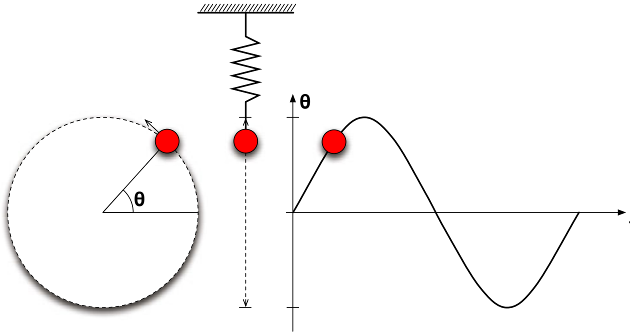

Fig. 2. Periodic behaviour of the simple harmonic oscillator

The reason may be found in the fact that the models that give rise to these signatures describe what will happen (or what could happen, if there is a choice) to individual instances of classes of molecules or molecular complexes, but do not distinguish between instances of the same class. In simulations of the synchronized Krebs cycle and the p53-mdm2 pathway under conditions that promote sustained oscillations, indistinguishable instances of particular classes, such as citrate and active p53, appear and disappear periodically.

What could be the computational significance of such periodic structures? We see at least two possibilities.

Cyclic interpretation. If a element g of a permutation group G is interpreted as a function $g : X \to X$ from a subset X of the state set Q to itself, then if g is non-trivial, then some element $x \in X$ is actually moved by g, so when the system reaches state $x \in X$ and is acted upon by g it will transition to state $y \in X$, i.e. $xg = y$ (where the group action is taken as multiplication on the right). Now, if g is applied again, this time to y, another state $z \in X$ will be reached. This process could be repeated such that

$$x \to xg \to xg^2 \to xg^3 \to \cdots \to xg^{(n-1)} \to x, \qquad (3)$$

[4] We hasten to add that the permutation groups found in these two systems are *not* of the same kind. Therefore, the two systems are different in some important way. However, our first goal is to find a plausible explanation for the presence of permutation groups in the first place, and it is only in this specific sense that we mean that these two systems have the same algebraic signature.

where the arrow "→" means "multiply on the right by g", for some $n > 1$, where n divides the order of g.[5] This is periodic behaviour, or a cycle of the system's states, that the system can potentially exhibit.

Parallel interpretation. Although we are still far from pinpointing the characteristics of 'cellular computation', as discussed in [6] and consistently with the discussion in Section 2 there is one aspect that is hard to overlook: parallelism. It is obvious that biochemical events take place in parallel in solution. We also know that a group element acting on a set can be conceived of as permuting *all* the elements of that set, at once. In the language of automata, this corresponds to the parallel transition of *every* state of the automaton to its next state as determined by the transition function, in parallel. This is not a single machine, clearly. We are talking about an ensemble of machines that are computing in parallel, each of which is at a different location along the *same* algorithm.

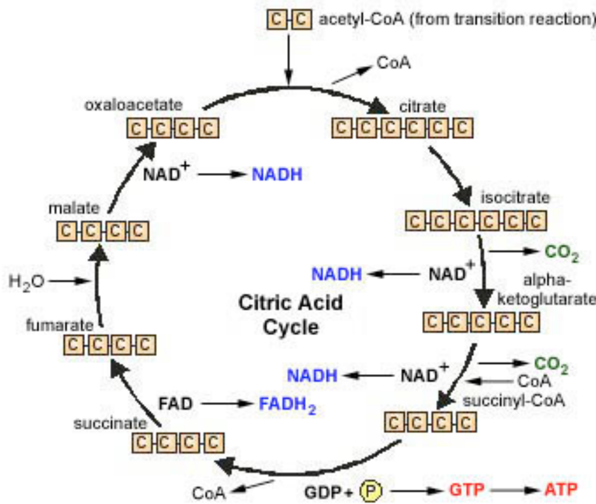

Fig. 3. Schematic of the Krebs or citric acid cycle [17]

A specific example could help elucidate the above point. Take, for instance, the Krebs or citric acid cycle, shown in Figure 3. It is clear that all the reactions around the cycle can be happening simultaneously and in parallel. But if we assume that the average concentrations of the metabolites in each stage around the cycle are constant then, strictly speaking, this system is always in the same state. Thus, in this case the discretization of this system into a finite-state automaton must be based on the history of an individual molecule as it travels around the circle. Such a molecule will indeed experience different 'states' as it is transformed at each stage. The hierarchical

[5] For example, $g = (123)(45)$ has order 6, and g acting repeatedly on the state 4 is $4 \rightarrow 4g = 5 \rightarrow 5g = 4g^2 = 4$, i.e. $n = 2$ in this case and 2 divides 6.

decomposition of the semigroup associated with such an automaton contains non-trivial permutation groups, e.g. C_2 and C_3 [9, 22]. This could indicate the parallel transition of 2 and 3 (abstracted or 'macro-') states, respectively under the same metabolic program (sequence of atomic transition operations). Once the concept of parallel transitions has been accepted as a possibility the generalization of the same mechanism to the simultaneous transition of all the states and macro-states permuted by a given subgroup, as we are proposing here, is equally possible and worth investigating further.

3.3 A Conceptual Architecture for the IM

Building on the above discussion, let us now simplify the system such that it is closed and let us neglect the irreversible (semigroup) components for the moment. Rather than performing an isolated algorithm to evaluate a mathematical function, as a Turing machine does, we are talking about an 'open' or 'permeable' algorithm, which can be coupled to other algorithms at each of its steps. A non-trivial question concerns the form such a coupling could or should take.

As we saw in Section 2 an algorithm is realized by one or more generating semigroup elements concatenated in a sequence or string of symbols. Because each semigroup element is a mapping $Q \to Q$, knowledge of the algorithm and of the state space implies knowledge of the state transition function λ. Therefore, an algorithm is a part of the realization of a sequential machine or mathematical function in the sense of Rhodes. It may therefore be more useful and appropriate to talk about the coupling between automata rather than between algorithms.

Automata can be coupled in different ways, for example by overlapping on one or more states. Such a case seems important for modelling symbiosis but presents some difficulties, since a common state for two automata would seem to imply that the two automata are actually associated with the same physical system and simply drawing on different sets of its states, with one in common. Alternatively, the output alphabet of an automaton could have some symbols in common with the input alphabet of another automaton, and vice versa. In this way, each automaton reaching certain states would trigger inputs to the other, causing it to transition. The same arrangement could be generalized to 3 or more automata coupled in different ways. Thus, state transitions taking place in one automaton could trickle through the system, causing other automata to advance their own algorithms.

What remains to be seen is whether such an architecture of computation will yield more powerful qualities (in the sense of non-functional requirements), e.g. greater efficiency, where efficiency could be defined as

$$\frac{\text{number of functional requirements satisfied}}{\text{number of state transitions required}} \tag{4}$$

This is in fact what the map of metabolic pathways looks like: a complex network of automata intersecting or communicating at several steps, which therefore means that the chemical reactions taking place at each of these intersection points are 'serving' multiple algorithms at the same time. This is one of the points of inspiration of the concepts of interaction computing, symbiotic computing, and multifunctionality [7].

From an evolutionary point of view, the emergence of an architecture that seems to violate just about all the accepted principles of software engineering (modularization, separation of concerns, loose coupling, encapsulation, well-defined interfaces, etc) might be explainable simply in terms of efficiency. In the presence of finite energy resources and multiple survival requirements, the organisms and the systems that survived were those that could optimize the fulfilment of such requirements for the most economical use of materials and energy; hence the overloading, the multifunctionality, and so forth. This in fact is not an unfamiliar concept in software engineering: where resources are constrained, for example in specialized drivers for real-time systems, it is common to code directly in assembly language and to develop code that is incredibly intricate and practically impossible for anyone but the engineer who wrote it to understand. Evolution has achieved the same effect with biological systems, but to an immensely greater extent.

Whether or not the above concepts will withstand closer scrutiny as we continue to develop the theory, it is interesting to show a visualization of the interaction machine according to Rhodes, in Figure 4 ([22]: 198). Rhodes does not use this term, but he does talk about the DNA and the cytoplasmic processes as interacting automata. His description of the cell as a system of interacting automata comes after the formal development of the sequential machine and its realization (that we presented above) all the way to the Krohn-Rhodes prime decomposition theorem ([22]: 62), and after a lengthy analysis and discussion of the Krebs cycle. Therefore, we expect Rhodes's work to continue to provide valuable inputs to our emerging theory of interaction computing.

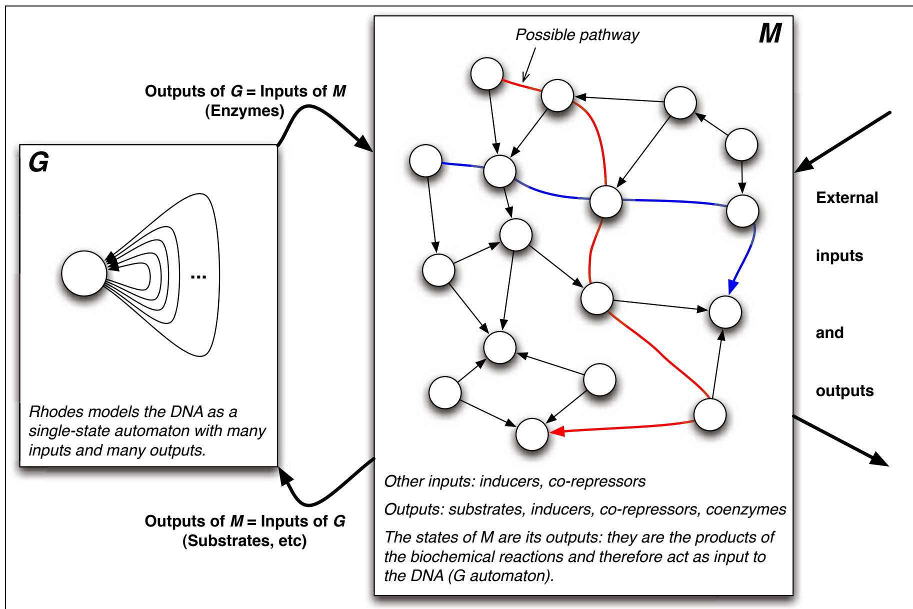

Fig. 4. Conceptual architecture of the biological realization of the Interaction Machine, based on Rhodes ([22]: 198)

4 Some Results from the Algebraic Analysis of the p53-mdm2 Regulatory Pathway

By using the now available SgpDec computer algebra software package [12] we can start applying these algebraic methods to real-world biological problems.

4.1 Petri Net of p53-mdm2 System

Here we briefly study the algebraic structure of the p53-mdm2 system. The p53 protein is linked to many other proteins and processes in the cell, but its coupling to the mdm2 protein appears to be particularly important for understanding cancer. Depending on the concentration level of p53, the cell can (in order of increasing concentration): (1) operate normally; (2) stop all functions to allow DNA repair to take place; (3) induce reproductive senescence (disable cellular reproduction); and (4) induce apoptosis instantly (cell ‘suicide’). Therefore, p53 is a very powerful and potentially very dangerous chemical that humans (and most other animals) carry around in each cell, whose control must be tuned very finely indeed. Roughly 50% of all cancers result from the malfunction of the p53- mdm2 regulatory pathway in damaged cells that should have killed themselves.

P53 levels are controlled by a fast feedback mechanism in the form of the mdm2 protein. P53 is synthesized all the time, at a fairly fast rate; but the presence of p53 induces the synthesis of mdm2, which binds to p53 and causes it to be disintegrated. When the DNA is damaged (for instance by radiation in radiotherapy) the cell responds by binding an ATP molecule to each p53, bringing it to a higher energy level that prevents its destruction and causes its concentration to rise. Thus in the simplest possible model there are in all 4 biochemical species: p53 (P), mdm2 (M), p53-mdm2 (C), and p53* (R), whose concentrations are modelled by 4 coupled and non-linear ordinary differential equations (ODEs) [24]. Whereas [16] analyzes the algebraic structure of this pathway from the point of view of the Lie group analysis of the ODEs, in this paper we look at the algebraic structure of the same pathway from the point of view of the discrete finite-state automata that can be derived from it.

4.2 Algebraic Decomposition

It is now a common practice to model biological networks as Petri nets. Petri nets can easily be transformed into finite automata, though this operation should be carried out with some care (for a detailed explanation see [11]). Depending on the capacity allowed in the places of the Petri net, the automaton’s state set can be different in size, thus we can get different resolutions during discretization.

When we distinguish only between the presence and absence (in sufficient concentration) of the molecules, the derived automaton has only 16 states (see Fig. 5).[6] This may be considered as a very rough approximation of the original process, but still group components do appear in this simple model: S_3 , the symmetric group on 3 points is among the components. The existence of these group components is clearly connected to oscillatory behaviours, however the exact nature of this relation still needs to be investigated.

[6] 4 places each taking on 2 possible values (0 or 1) makes $2^4 = 16$ states.

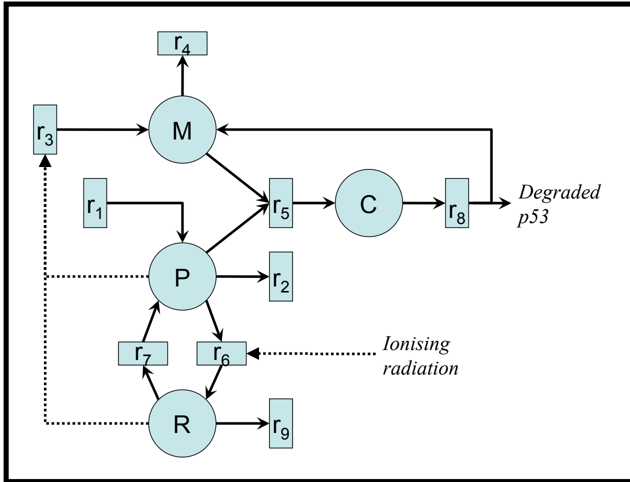

Fig. 5. Petri net for the p53-mdm2 regulatory pathway [10]. P = p53, M = mdm2, C = p53-mdm2, R = p53*.

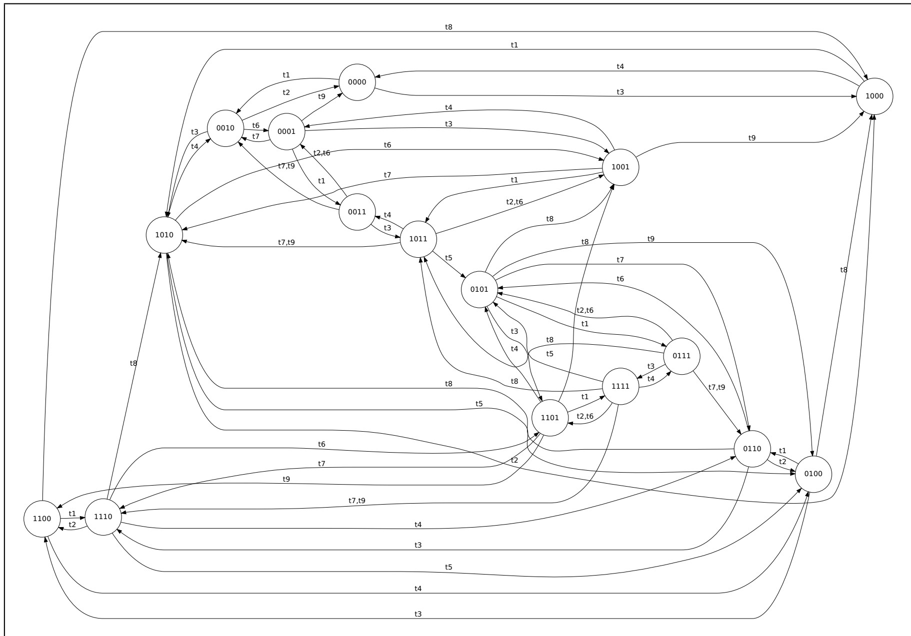

Fig. 6. Automaton derived from 2-level Petri net of the p53 system (16 states). The labels on the nodes encode the possible configurations for M, C, P and R (in this order). 0 denotes the absence, 1 the presence of the given molecule. For instance, 0101 means that C and R are present.

If we in addition distinguish two levels of significant concentration, i.e. absence (or very low, sub-threshold concentration), low, and high levels concentrations, then the corresponding automaton has 81 states ($3^4 = 81$) and the algebraic decomposition reveals that it contains S_5 among its components. The appearance of S_5 is particularly interesting as it contains A_5 , the alternating group on 5 points, which is the smallest SNAG (simple non-abelian group). The SNAGs are considered to be related to error-correction [22] and functional completeness [18, 15, 7]. The latter property of SNAGs makes them a natural candidate for realizing an analogue of 'universal computation' within the finite realm. The presence of a SNAG in the decompostion of the p53-mdm2 path- way suggests that the cell could potentially be performing arbitrarily complex finitary computation using this pathway. Moreover, how the activity of the rest of the cell interacts with the p53-mdm2 pathway to achieve such fine regulatory control could depend on other components (interacting machines) in the interaction architecture of the cell harbouring similar group structures.

5 Conclusions

In this paper we have tried to paint a broad-brush picture of the possibilities for the roles of algebraic structures in enabling open-ended constructive computational systems, based on recent work and on our current results. The most immediate example is to regard a transformation semigroup as a constructive dynamical space in which computation can be carried out, analogous to programming languages on present-day computers. Although practical implementations of automata in real computers cannot help being finite, the size of the state space of a modern computer is well beyond our human ability to count without losing all sense of scale (many times the number of molecules in the known universe) and so 'approximates' the infinite size of a Turing machine reasonably well for practical purposes.

The work of Rhodes suggests, however, a much stronger claim, that sequential and interacting machines inspired by biological systems must be finite in a much more limiting sense, and that algebraic structure is the key to finitary computation and to understanding how computation can proceed via interaction. This leads us to a meta-space of constructive dynamical systems in which the interacting parts that synergetically conspire to achieve complex computation are individual machines or transformation semigroups themselves. We are combining this insight with the results of the algebraic analysis of the semigroups derived from cellular pathways and with more synthetic tentative arguments about the architecture of an Interaction Machine that could replicate biological computation. We have not yet reached actionable conclusions, but we feel we are uncovering interesting mathematical and computational properties of cellular pathways and are finding intriguing correspondences between biochemical systems, dynamical systems, algebraic systems, and computational systems.

Acknowledgments

Partial support for this work by the OPAALS (FP6-034824) and the BIONETS (FP6-027748) EU projects is gratefully acknowledged.

References

1. Ashby, W.R.: An Introduction to Cybernetics. Chapman & Hall Ltd., London (1956), http://pespmci.vub.ac.be/books/IntroCyb.pdf
2. Banzhaf, W.: Artificial chemistries towards constructive dynamical systems. Solid State Phenomena, 97–98, 43–50 (2004)
3. Borges, J.L.: La biblioteca de Babel (The Library of Babel). El Jardin de senderos que se bifurcan. Editorial Sur. (1941)
4. Briscoe, G., Dini, P.: Towards Autopoietic Computing. In: Proceedings of the 3rd OPAALS International Conference, Aracaju, Sergipe, Brazil, March 22-23 (2010)
5. Dini, P., Briscoe, G., Van Leeuwen, I., Munro, A.J., Lain, S.: D1.3: Biological Design Patterns of Autopoietic Behaviour in Digital Ecosystems. OPAALS Deliverable, European Commission (2009), http://files.opaals.org/OPAALS/Year_3_Deliverables/WP01/
6. Dini, P., Horvath, G., Schreckling, D., Pfeffer, H.: D2.2.9: Mathematical Framework for Interaction Computing with Applications to Security and Service Choreography. BIONETS Deliverable, European Commission (2009), http://www.bionets.eu
7. Dini, P., Schreckling, D.: A Research Framework for Interaction Computing. In: Proceedings of the 3rd OPAALS International Conference, Aracaju, Sergipe, Brazil, March 22-23 (2010)
8. Dini, P., Schreckling, D., Yamamoto, L.: D2.2.4: Evolution and Gene Expression in BIONETS: A Mathematical and Experimental Framework. BIONETS Deliverable, European Commission (2008), http://www.bionets.eu
9. Egri-Nagy, A., Nehaniv, C.L., Rhodes, J.L., Schilstra, M.J.: Automatic Analysis of Computation in BioChemical Reactions. BioSystems 94(1-2), 126–134 (2008)
10. Egri-Nagy, A., Nehaniv, C.L., Schilstra, M.J.: Symmetry groups in biological networks. In: Information Processing in Cells and Tissues, IPCAT'09 Conference, April 5-9 (2009) (Journal preprint)
11. Egri-Nagy, A., Nehaniv, C.L.: Algebraic properties of automata associated to petri nets and applications to computation in biological systems. BioSystems 94(1-2), 135–144 (2008)
12. Egri-Nagy, A., Nehaniv, C.L.: SgpDec - software package for hierarchical coordinatization of groups and semigroups, implemented in the GAP computer algebra system (2008), http://sgpdec.sf.net
13. Golding, D., Wegner, P.: The Church-Turing thesis: Breaking the myth. In: Computability in Europe (CiE) conference series (2005)
14. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2001)
15. Horvath, G.: Functions and Polynomials over Finite Groups from the Computational Perspective. The University of Hertfordshire, PhD Dissertation (2008)
16. Horvath, G., Dini, P.: Lie Group Analysis of p53-mdm3 Pathway. In: Proceedings of the 3rd OPAALS International Conference, Aracaju, Sergipe, Brazil, March 22-23 (2010)
17. ICT4US, http://ict4us.com/mnemonics/en_krebs.htm
18. Krohn, K., Maurer, W.D., Rhodes, J.: Realizing complex boolean functions with simple groups. Information and Control 9(2), 190–195 (1966)
19. Krohn, K., Rhodes, J.: Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. Transactions of the American Mathematical Society 116, 450–464 (1965)

20. Krohn, K., Rhodes, J.L., Tilson, B.R.: The prime decomposition theorem of the algebraic theory of machines. In: Arbib, M.A. (ed.) Algebraic Theory of Machines, Languages, and Semigroups, ch. 5, pp. 81–125. Academic Press, London (1968)
21. Lahti, J., Huusko, J., Miorandi, D., Bassbouss, L., Pfeffer, H., Dini, P., Horvath, G., Elaluf-Calderwood, S., Schreckling, D., Yamamoto, L.: D3.2.7: Autonomic Services within the BIONETS SerWorks Architecture. BIONETS Deliverable, European Commission (2009), `http://www.bionets.eu`
22. Rhodes, J.L.: Applications of Automata Theory and Algebra via the Mathematical Theory of Complexity to Biology, Physics, Psychology, Philosophy, and Games. World Scientific Press, Singapore (2009); foreword by Hirsch, M.W. edited by Nehaniv, C.L. (Original version: University of California at Berkeley, Mathematics Library, 1971)
23. Turing, A.: On Computable Numbers, with an Application to the Entschei-dungsproblem. Proceedings of the London Mathematical Society 42(2), 230–265 (1936); a correction. ibid 43, 544–546 (1937)
24. Van Leeuwen, I., Munro, A.J., Sanders, I., Staples, O., Lain, S.: Numerical and Experimental Analysis of the p53-mdm2 Regulatory Pathway. In: Proceedings of the 3rd OPAALS International Conference, Aracaju, Sergipe, Brazil, March 22-23 (2010)