2008 Second IEEE International Conference on Digital Ecosystems and Technologies (IEEE DEST 2008)
© 2008 IEEE.

Notes on Abstract Algebra and Logic: Towards their Application to Cell Biology and Security

*Paolo DINI and †Daniel SCHRECKLING

*Department of Media and Communications, London School of Economics and Political Science,
London, United Kingdom, e-mail: p.dini@lse.ac.uk,
†Security in Distributed Systems, University of Hamburg, Hamburg, Germany,
e-mail: schreckling@informatik.uni-hamburg.de

Abstract— **This paper begins to chart and critically analyse the formal connections between algebra, logic, and cell biology on the one hand, and algebra, logic, and software security on the other. Much of the discussion is necessarily conceptual. Where the discussion is more formal the current distance between these disciplines appears evident. The paper focuses on the algebra of network coding, reviews the main types of algebraic and temporal logics that underpin security, and briefly discusses recent work in the application of algebra and logic to the DNA code.**

Index Terms— **cell biology, abstract algebra, algebraic logic, network coding, software security**

I. Introduction

In this article we wish to begin to discuss some ideas related to the connections between cell biology and software security. The bridge that we are in the process of building between these two very different fields relies on abstract algebra and logic and can be simplistically depicted as follows: biology $\rightarrow$ algebra $\rightarrow$ logic $\rightarrow$ security. In this context security represents one of the possible applications of a formalism that we expect to be of wider relevance. When referring to biologically inspired computing, reliance on some kind of evolutionary framework tends to be assumed by default. Whereas biological evolution does represent an essential model for biologically inspired computing, in this paper we are focussing on the 'other' function of DNA. By this we mean all the processes relating to the life of the individual organism, thus a better name could be 'development', or 'morphogenesis', or 'gene expression'. As Stuart Kauffman says,

> Darwin's answer to the sources of the order we see all around us is overwhelmingly an appeal to a single singular force: natural selection. It is this single-force view which I believe to be inadequate, for it fails to notice, fails to stress, fails to incorporate the possibility that simple and complex systems exhibit order spontaneously. [1]

If evolution is the model that is able to explain phylogeny (a succession of organic forms sequentially generated by reproductive relationships), in this paper we are addressing the construction of a model that may eventually be relevant to ontogeny (the history of structural changes in a particular living being) [2]. Before we can begin to understand and model morphogenesis, however, we need to understand gene expression, on which morphogenesis depends. The biology part of this paper is therefore focussed entirely on the cell. Our objective is in fact to develop a model inspired by cell metabolic processes that can represent equally well biological and computing processes. The motivation for studying gene expression over evolution is that, even though it too is the result of an evolutionary process, it is a much faster and more powerful process of self-organisation that does not rely at all on genetic operators, but on rather different mechanisms.

As discussed in Section IV and in [2-5], although we can make the mapping from cell metabolic cycles to digital algorithms seem plausible, we still face the problem of the absence of physical interaction forces between digital entities, and of the concept of temperature. In other words, we cannot rely on the minimisation of free energy as the driver of software systems. More importantly, the interaction forces bring with them a built-in regularity that is a direct consequence of the regularity and universality of physical laws and that, it seems plausible to conclude, gives rise to the observed regularities in structure and behaviour of biological systems. Our research is based on the assumption that the regularities that result from the interplay between physical law and the spatio-geometrical characteristics of matter can be formalised through the mathematical theory of groups, rings, and fields, which is a branch of abstract algebra. Parallel work by [3] supports this perspective and is in fact ahead of our results. If we then identify the flow of energy with a flow of information we do not really need to worry about the lack of interaction forces. The behaviour of the users of the software and communication system will provide a constant flow of information which, in our view, can be constrained by algebraic transformation and interaction rules to produce ordered structures and behaviour. In reference to Fig. 1, we can therefore see why there is an arrow between algebra and cell biology. The importance and effectiveness of abstract algebra, symmetries, fields, and groups is demonstrated well by network coding techniques, which will therefore be discussed in this paper as a useful example of the theory.

The arrow from cell biology to Interaction Computing is more difficult to explain, partly because the concept of interaction computing is still being formed [4] [5]. In order for the scenario described above to function, the digital system needs to become *reactive* to the inputs and the behaviour of the users. In other words, there cannot be computation without interaction. Iterating this concept recursively to components that are farther removed from the user interface, they too cannot change their states without being 'pushed'

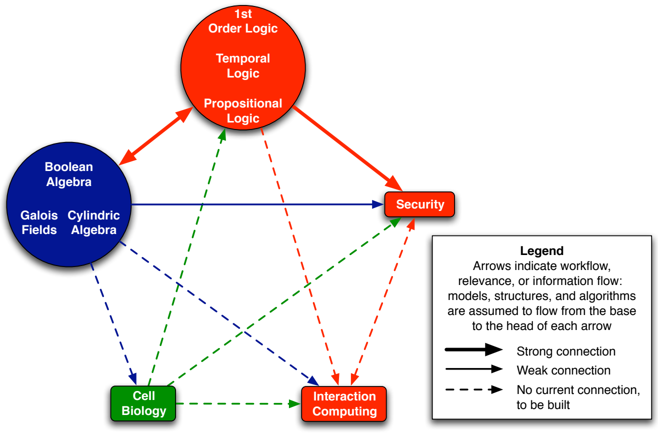
1st Order Logic
Temporal Logic
Propositional Logic
Boolean Algebra
Galois Fields
Cylindric Algebra
Security
Cell Biology
Interaction Computing
Legend
Arrows indicate workflow, relevance, or information flow: models, structures, and algorithms are assumed to flow from the base to the head of each arrow
Strong connection
Weak connection
No current connection, to be built

Fig. 1. Disciplinary connections of relevance to bio-inspired security

by the components that precede them. The picture that emerges can therefore be characterised conceptually as a set of coupled and interacting finite state machines whose state spaces are subdivided into permissible regions bounded by surfaces defined by algebraic structure laws. Each state machine is performing a piece of the algorithm. This is thus what we mean by interaction computing for the execution of a distributed algorithm that is partly resident in the 'DNA' of the digital system and partly in the environment.

Since Boole published his short book on the mathematics of logic in 1847 [6], algebra and logic have been closely connected. The algebraisation of logic has been one of the success stories of the 20th Century [7] [8] [9]. Therefore, we do not expect to say anything fundamentally new in this regard, in this paper. However, if we take the interface between algebra and logic to be the 'centre of gravity' of this article, we are interested in shedding some light in two different directions. On the one hand, in Section III we will review the mapping of propositional logic to Boolean algebra, briefly explain the algebraisation of First-Order Logic, and finally show the link between First-Order and Temporal Logic, which is widely used in the security domain to support applications that are able to describe, detect, and prevent specific security features or breaches, respectively. On the other hand, after reviewing the fundamentals of abstract algebra, touching on coding and Galois theory, in Section IV we will discuss briefly how physical and evolutionary forces have induced an algebraic structure on the DNA code. We see this as the first step in the charting of a possible path from order construction in biology to autonomic security algorithms and data structures that weaves its way through algebra and logic.

II. Abstract Algebra

Abstract algebra deals with the operations between the elements of a set, with the mappings between different sets, and with mappings defined from a set to itself. The elements of a set can be anything at all, including other sets. They can be integers, polynomials, functions, 'variables', and so forth. They can also be the constitutive elements of logic, which provides one of the fundamental motivations for the line of investigation pursued in this paper. Finally, we are making an explicit assignment of some yet unspecified elements of the cell as elements of a set. The temptation is to treat the cell itself as a set. Although this is not necessarily wrong, we are starting with much simpler sets such as, for example, treating the four bases of the DNA as the elements of a set that we can call the alphabet of DNA. The rest of this section relies on [10], where the basics of abstract algebra are reviewed and summarised in support of this discussion.

A. Applications to Network Coding

Our purpose is not to summarise results that have been known for 40 years or more in order to apply them in the manner originally intended. For that, we need only provide bibliographical references. The point of developing all this algebra, from the point of view of our research in bio-inspired computing, is not functionalist (i.e. to improve performance); it is to understand the algebraic structure itself and then see what this structure has in common with logic on the one hand and with the DNA on the other, as will be discussed in subsequent sections of this paper. There is of course an expectation that the biological insights will ultimately bring a number of advantages, including better performance, but the first objective is to develop a common formalism for these very different domains. To this end, network coding is a very useful 'case study' because it provides a very practical context against which the abstract concepts discussed in [10] will hopefully become easier to understand. The following discussion relies on [11] and [12], and to a smaller extent on [13].

We need to introduce some basic terminology of linear block codes, upon which we will build a powerful algebraic structure by progressively introducing additional constraints. We start with the notion of a vector space. Put simply (even if perhaps too simplistically), a **vector space** is a field in more than one dimension. Each element of a vector space of dimension n is a tuple of n objects, with each object an element of a field F. An n-dimensional vector space over F is denoted by F^n. If F is the binary field $\mathbb{Z}_2$, F^n has exactly 2^n elements. Linear block codes operate by adding redundancy to a message in order to allow the original message to be reconstructed in the presence of transmission errors. Given a signal stream, we break it up into an unspecified number of message blocks of length k, taken from F^k. Thus, the message space is a vector space of dimension k and exactly 2^k elements (message blocks). To make the code unambiguous and invertible back to the original message, the mapping between the message space and the code space must be 1-1 and invertible. Therefore, the code space is a k-dimensional sub-space of F^n and also contains 2^k different elements, although each is of length $n > k$. The positive consequence of setting things up in this manner is that the encoding map becomes a linear transformation (i.e. a matrix multiplication) ([12], 239). Thus, we multiply each message block (row vector) by a matrix of dimensions $(k \times n)$ (k rows by n columns). The result of this multiplication for each block is a code vector of length n, called a codeword. Each codeword is sent wirelessly by the transmitter and received by the receiver, possibly with some errors. The receiver must detect these errors, correct them, and then map each codeword back to the original message block to reconstruct the original signal.

We now start connecting to abstract algebra by noticing that in network coding applications the focus of attention is not a polynomial whose roots we need to find through the generation of a field extension as a factor ring (Galois's original problem) but, rather, the factor ring itself. Furthermore, the generation of a field extension is a necessary step in finding the roots of an *irreducible* polynomial. As it happens a polynomial does not need to be irreducible to form an ideal, and therefore it does not need to be irreducible to form a factor ring. An additional point that should help connect the work in [10] to this discussion is that the concept of ideal can be applied, in a sense, recursively. In other words, a factor ring is obtained by 'dividing' a ring $GF(2)[x]$ by an ideal, say $[(x^n - 1)]$. The resulting factor ring can also contain ideals. It is these latter ideals that we are mainly going to use. More precisely, in the discussion of cyclic codes that follows we identify the vector space F^n with the factor ring, and the code space with one of its ideals.

A **cyclic code** is a linear block code with the characteristic that every cyclic shift of any of its codewords yields another codeword. A **cyclic shift** is the n-tuple obtained by shifting every component to the right (or left) and wrapping the last component back to the front of the list. When working with cyclic codes, it is useful to associate each codeword of length n with a polynomial of degree $n-1$ or less, whose coefficients are the coordinates of the code vector:

$$c(x) = c_0 + c_1x + c_2x^2 + ... + c_{n-1}x^{n-1}.$$

$c(x) \in F[x]$ is called the code polynomial. The set of k code polynomials of a code C is a subset of the factor ring $GF(2)[x]/[(x^n - 1)]$, which is not a field since $(x^n - 1)$ is factorisable. Incidentally, from this fact we deduce that whereas every finite field contains a number of elements that is equal to an integer power of a prime number, the converse is not true: a set that contains a number of elements that is equal to an integer power of a prime number is not necessarily a field. To clarify, the factor ring $GF(2)[x]/[(x^n - 1)]$ is not a field because not everyone of its elements has an inverse (due to the fact that the GCD of each element and $(x^n - 1)$ is not necessarily 1). From this point of view the factor ring is a one-dimensional object. But the same object can also be described as a 'vector field' of dimension n, F^n. From this second point of view the use of the term 'field' refers to the fact that the *ground* field $0, 1$ is indeed a field. We notice that in this factor ring, by construction, operations between its elements are performed modulo $(x^n - 1)$. The reason for creating the factor ring using $(x^n - 1)$, rather than an irreducible polynomial of the same degree, is that in the latter case the factor ring would be a field, and therefore its only ideals would be 0 and itself. Why this matters is shown next.

In the polynomial ring $GF(2)[x]/[(x^n - 1)]$ the cyclic shift of a codeword by k is equivalent to multiplication modulo $(x^n - 1)$ of the corresponding code polynomial by x^k. This fact in itself seems unremarkable until we realise that an arbitrary polynomial $a(x)$ can be considered as a linear combination of cyclic shifts which, according to the definition, yields another $c(x)$ that still belongs to the same code C. Because we already knew that C is a sub-space of F^n (or a subring of the factor ring), we know that it is closed under addition: adding two vectors yields a third vector in the same plane the two vectors define, even if this plane is immersed in a 3-D space. As a consequence, the code C satisfies the ideal test and is in fact an ideal of the ring $GF(2)[x]/[(x^n - 1)]$. Because it is a subset, the Hamming distance between codewords is greater than 0, which is what makes error correction possible.

The ring of polynomials $GF(2)[x]$ is a Principal Ideal Domain, which means that every one of its ideals is principal. In our case we are working with $(x^n - 1)$. The factor ring $GF(2)[x]/[(x^n - 1)]$ is not a field and is not a PID either. However, every one of its ideals *is* principal ([12], 245). This implies that our code C is entirely generated by a single polynomial $g(x)$, aptly called the **generator polynomial**. The generator polynomial is the monic polynomial of least degree (for a particular code) that belongs to C and from which every element of C can be generated through multiplication by elements of $GF(2)[x]$ modulo $(x^n - 1)$. There is a unique such polynomial for any ideal of $GF(2)[x]/[(x^n - 1)]$, and each such instance divides $(x^n - 1)$ ([13], 32). It follows that to generate all the possible cyclic codes for a given value of n we need to find all the irreducible factors of $(x^n - 1)$. The possible generator polynomials $g(x)$ are all the possible divisors of $(x^n - 1)$, formed as single irreducible factors or as products of these irreducible factors. If $g(x)$ is chosed of degree $n - k$, a linear code results and the generator matrix can be formed starting from the simple statement $m(x)g(x) = c(x)$, where $m(x)$ is a polynomial of degree $k - 1$ representing a message block. Furthermore, there exists a polynomial $h(x)$ such that $g(x)h(x) = (x^n - 1)$, which gives rise to the check matrix.

In building cyclic codes it is easy to set their length and dimension. However, the spacing (in terms of Hamming distance) of the $c(x)$ codewords within the vector space F^n is not necessarily uniform, which means that the minimum distance is both uncertain and not easy to find, especially as n becomes large. BCH codes are also cyclic codes, but they introduce an additional constraint that makes it possible to specify the minimum distance d at the outset. The fact that, as already mentioned, the vector field F^n has the same cardinality as $GF(2^n)$ may at first seem to imply that F^n is a field, which is potentially confusing since we just spent considerable effort to prove that F^n (in the present context) is a ring and not a field. Isomorphism, however, requires more than cardinality. Because it is a vector field of *remainders*, it cannot in fact be considered independently of the ideal that generated it as a factor ring, a fact that becomes evident when multiplying elements modulo different ideals.

For BCH codes the generator polynomial is the least common multiple of the minimal polynomials of $d - 1$ consecutive powers of a primitive n^{th} root of unity, for a desired minimum distance d. A **primitive** n^{th} root of unity in a field F is an element a whose order in the multiplicative group of F is precisely n. As a consequence, a and all of its powers are roots of the equation $x^n - 1 = 0$. By the definition of order of a group element, all the powers of a up to a^{n-1} are distinct. Therefore, all the n roots of $x^n - 1 = 0$ are generated by a and its powers. From these we can choose $d - 1$ consecutive ones.

For a given code length n we need to find a primitive n^{th} root of unity, and the smallest field that contains $GF(q)$ and a primitive n^{th} root of unity is $GF(q^e)$, where e is the order of q mod n ([12], 248). If n and q are two coprime integers, the **order** of q mod n is the smallest positive integer e for which $q^e = 1$ mod n. In other words, e is the order of q in the multiplicative group $\mathbb{Z}_n$. For example, if $n = 10$ and $q = 3$, $e = 4$. In other words, the smallest field that contains a primitive 10^{th} root of unity over the base field $GF(3)$ is $GF(3^4)$. As another example, for $GF(2)$ and $n = 21$ $e = 6$, because $2^6 = 64 = 1 \mod 21$. In practice, one tends to use a value of $n = 2^e - 1$: $n = 127$ for $e = 7$, or $n = 255$ for $e = 8$. These codes are called, unsurprisingly, primitive BCH codes since the n^{th} root of unity β in such a case is a primitive element of $GF(2^e)$.

B. An Observation on Gordon's Method

Gordon's method [14] is relevant to BCH codes and is concerned with finding the minimum polynomials of the roots in an extension field $GF(2^e)$. These roots can be expressed as consecutive powers of a primitive element or as the remainders of $f(x)$ in the factor ring $GF(2)[x]/[f(x)]$, where $f(x)$ is the irreducible primitive polynomial that generates the extension field $GF(2^e)$. Because the degree of these minimum polynomials can be at most e, there are several in any given extension field with 2^e roots. Gordon realised that, for the same reason, each minimum polynomial can also be expressed modulo the same primitive polynomial. In other words, each minimum polynomial can be expressed as one of the elements from the set of its own roots. This makes the determination of the minimal polynomials trivially simple and ideally suited for low-power space probes, where every bit of memory and CPU cycle counts. Even more surprisingly, we cannot avoid the conclusion that a minimum polynomial and one of its roots may actually have the same polynomial form! It is not clear whether this fact has any significance, but such examples of structural invariance cannot help but stimulate our curiosity. This particular example may be explainable by digging deeper into Galois theory, but is reminiscent of conformal invariance of differential equations under the action of a Lie group of transformations or, to a smaller extent, of the concepts of eigenvector in oscillatory systems or the renormalisation group of statistical physics.

In this section we have used network coding to give a flavour for the richness of algebraic structure, which may at first seem quite far removed from biology. As it happens, the work reported in [3] indicates otherwise. But let's turn to the structure of logic first.

III. Logic, Algebra, and Security

Having presented a general overview of abstract algebra and its very basic concepts, this section is going to establish a link between the disciplines of logic, algebra, and security. For this purpose we give a short introduction to propositional and first-order logic (FOL). We show how the different logics find their counterparts in the realm of abstract algebra. With the informal extension of the basic concepts of logic and algebra to temporal logic and quantifier algebras, respectively, we will finally bridge from algebra to security.

A. Propositional and First-Order Logic

In this section we want to give a short introduction to propositional and predicate logic, two important fields of the research area of symbolic logic. Of course, this introduction cannot be exhaustive and we will refer the reader to appropriate literature.

Propositional logic (PL) is often called sentential logic. Both names account for the nature of this logic since it uses sentences, thus *sentential*, which can be thought of as *propositions*. Classical PL studies the logical values of sentences and the effects of propositional connectives, such as *and* and *or*, which can be used to form new sentences out of atomic sentences.

Very basic building blocks of sentences in PL are so called *atomic formulas*. As PL studies classical logical values two atomic formulas explicitly denote the truth values "true" and "false". To represent these values in propositional formulas the symbols $\top$ and $\bot$ are used. The interpretation of these symbols is defined using a so-called Boolean valuation function v. It maps atomic formulas to a set $\mathcal{T} = \mathbb{Z}_2$ (see Section II-A) and defines $v(\bot) = 0$ and $v(\top) = 1$. Additionally, PL knows another type of atomic formulas: propositional letters. They represent the truth values of propositions and are therefore sometimes called variables. Their truth values are also defined by the valuation function.

To construct formulas with these atomic formulas, PL also defines operators, negation ($\neg$) and disjunction ($\vee$). According to the following rules, they can be used to construct the set $\mathcal{P}$ of propositional formulas:

1) If A is an atomic formula, $A \in \mathcal{P}$,
2) If $X \in \mathcal{P}$ then $\neg X \in \mathcal{P}$, and
3) If $X, Y \in \mathcal{P}$ then $(X \vee Y) \in \mathcal{P}$.

In natural language $\vee$ is often denoted by the word "or", i.e. if one of the propositions combined by $\vee$ is "true", the combination of the propositions is also "true". The natural interpretation of negation is the inversion of the truth value from "true" to "false" and vice versa. As we implicitly associated the truth value "true" with 1 we can easily extend the above valuation function to:

1) $v(\top) = 1; v(\bot) = 0$
2) $v(\neg X) = 1 - v(X)$ with $X \in \mathcal{P}$
3) $v(X \vee Y) = max(v(X), v(Y))$ with $X, Y \in \mathcal{P}$

It can be shown that the two operators suffice to express all possible logical binary operators. As this is burdensome PL uses abbreviations to express these additional operators. As we consider binary connectives and because our input valuation function maps formulas to a two element set, namely $\mathbb{Z}_2$, there are $2^{2^2} = 16$ such possible abbreviations.

The valuation of propositional formulas and the evaluation of polynomials are both important and help us relate the theoretical concepts to observable or experiential reality. However, as will be discussed below in Section III-B, our concern is less with the kind of value the elements of our set can assume, but more with the structural relationships within and between sets. For example, the definition of operators in PL seems different from the definition of an algebraic field. However, it turns out that the same definition is more concisely expressed when using the operators $\wedge$ (multiplication mod 2) and $\not\equiv$ (addition mod 2). Similarly, the possible 16 operators or abbreviations introduced for convenience are analogous in logic to a field extension $GF(2^4)$ over the base field $GF(2)$.

As we have seen above, PL can be used to build formulas which represent propositions. They can be evaluated using Boolean valuation. However, PL is unable to derive valid arguments with respect to the internal structure of a proposition. For this reason *first-order logic* replaces the pure atomic formulas by predicates which can have arguments. Thus, FOL is often called predicate logic. Due to the introduction of variables FOL also supports quantifiers to bind variables.

FOL is defined over a first-order language $L(\mathcal{R},\mathcal{F},\mathcal{C})$. $\mathcal{R}$ is a finite, countable set of relation symbols (predicate symbols) with specific arity. The finite or countable set $\mathcal{F}$ contains function symbols each of which also has associated an arity. Finally, $\mathcal{C}$ is also a finite countable set of constant symbols. Based on $L(\mathcal{R},\mathcal{F},\mathcal{C})$, FOL defines *terms* over *variables* and, based on these terms, *formulas*. The definition of the latter is very similar to the definition of formulas in PL. The main difference is the use of terms instead of propositions and the introduction of universal ($\forall$) and existential ($\exists$) quantifiers.

Compared to PL the definition of FOL semantics is complicated as we are facing variables, functions, relations, and quantifiers. It uses a model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ for the first-order language $L(\mathcal{R},\mathcal{F},\mathcal{C})$ where $\mathcal{D}$ is a non-empty set, the so-called domain of $\mathcal{M}$. $\mathcal{I}$ is the interpretation of $\mathcal{M}$ which maps the elements of the language, i.e. constant, relation, and function symbols, to their algebraic counterparts defined over the domain $\mathcal{D}$. Using these mappings, it is possible to also define a recursive mapping for terms. As this definition associates each term with a value in $\mathcal{D}$ we arrive at the point where it is possible to associate a truth value with each formula defined in $L(\mathcal{R},\mathcal{F},\mathcal{C})$. This can be done in analogy to PL. The same propositional constants $\top$ and $\bot$, the same set $\mathcal{T}$, as well as the same connectives (denoted as $\circ$) are used for this purpose. As for PL it is also possible to introduce proof procedures which would show the full power of FOL. However, we skip these details and refer the interested reader to [15].

B. Algebraic Logic

Algebraic Logic is the field of research which deals with studies of algebras that are relevant for logic. It additionally investigates the methodology of solving problems in one of the two domains and of translating the solution back into its original domain. This section is going to show how propositional logic as well as FOL are connected to algebra.

In sub-section III-A we introduced the syntax and semantics of PL and described Boolean valuation. If we want to generalise the Boolean valuation and apply it to arbitrary formulas with n propositions we will have to investigate 2^n different *interpretations*. This characteristic becomes a problem if we want to find *logical consequences*.

In common language a logical consequence is a statement which follows from some other statements. In mathematics, for example, the set of statements could be axioms. So, if we want to *prove* in PL that a statement is a logical consequence of other propositional formulas one may use established proof procedures such as Hilbert systems or resolution. Logical consequences can also be interpreted as a means to find propositional formulas which are equivalent, i.e. which have the same model.

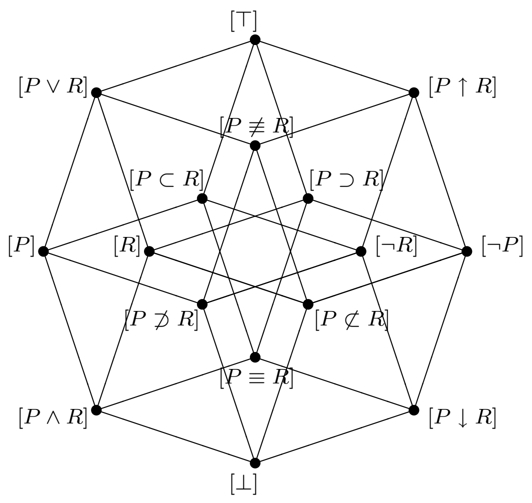

Fig. 2. Hasse diagram of Boolean lattice of PL generated by {P,R}

Due to the definition of PL any propositional formula has infinite equivalent formulas. However, it is possible to group all propositional formulas based on only two propositions P, R and obtain a set set $\mathcal{E}$ of 16 equivalence classes. According to the notion of *algebra*, we look at the interrelation between these equivalence classes. Among the 16 classes, we first find an equivalence class which only contains *tautologies*, i.e. for formula X, $v(X) = 1$ for every v. As $v(\top) = 1$ for all v, we call this equivalence class $[\top]$. Its counterpart is the equivalence class $[\bot]$.

Further, it is possible to define a partial order $\leq$ on $\mathcal{E}$.

This order is illustrated in Fig. 2 It shows a Hasse diagram, which is a graph representation for partially ordered sets (posets).

Two things can be recognised in this poset $(\mathcal{E}, \leq)$. First of all the poset is bound by a least and a greatest element, $[\bot]$ and $[\top]$ respectively. The second observation is not that obvious. Take representatives X and Y from two arbitrary equivalence classes $[X]$ and $[Y]$. The application of the propositional connectives $\vee/\wedge$ to X and Y yields a new propositional formula. This formula Z is a representative of an equivalence class $[Z]$. It is the supremum and the infimum respectively of $\{[X],[Y]\}$ and $[Z] \in \mathcal{E}$.

Here, we already see that the order-theoretic interpretation based on the partial order $\leq$ finds its equivalence in an algebraic structure based on the two operators $\vee$ and $\wedge$ and vice versa. Based on this observation and using the above Hasse diagram it can additionally be verified that any three elements of $\mathcal{E}$ also have the characteristics of associativity, commutativity, and absorption. Therefore, the partial order $(\mathcal{E}, \leq)$ is a lattice, which has interesting characteristics from an order-theoretic as well as from an universal algebra point of view. Additionally, we can show the distributivity of any three elements and the existence of the complement to each representative of an equivalence class. These five characteristics represent the axioms which have been set up by George Boole. Therefore, this distributive complemented lattice is also called a Boolean algebra.

Tarski and Lindenbaum were the first to precisely discuss the set of propositional formulas as an algebra with operators that were induced by the connectives of the propositional language. More details on the structural analysis are presented in [8], [16].

Summarising this section we can state that there is a strong link between PL and algebra. Thus, we are capable to choose a domain, PL or Boolean algebra, which offers the most suitable tools and the best knowledge to analyse a structure. Hence, it would be good if the same correspondence between FOL and algebra held true.

We proceed similarly and outline how Boolean algebra can be extended to obtain a FOL algebra. For this purpose we first introduce *quantifier algebras*. Their definition is very similar to the construction of a Boolean algebra out of PL. Thus, from the last section we simply collect the elements which we need for a formal definition.

Let Γ be the first-order language $L(\mathcal{R}, \mathcal{F}, \mathcal{C})$ and let $\langle v_\kappa \rangle_{\kappa<\alpha}$ be a sequence of variables. Let Θ be a theory of Γ. We define $\mathcal{F}^\Gamma$ as the set of all formulas of Γ. We currently have to restrict our considerations to the set of formulas that do not contain the formulas $F \equiv_\Theta G$. Here relation $\equiv_\Theta$ denotes the equality relation that can be deduced from Θ. Based on $L(\mathcal{R}, \mathcal{F}^\Gamma / \equiv_\Theta, \mathcal{C})$ and the general Boolean algebra $\mathcal{B} = \langle B, +, \cdot, -, \mathbf{0}, \mathbf{1} \rangle$ we can define the Boolean operations $+$ and $\cdot$ by mapping them to there counterparts in PL, $\vee$ and $\wedge$ respectively (also see Section III-A). $\mathbf{1}$ denotes all formulas of theory Θ. For simplicity and consistency with previous sections we write $\top$. $\mathbf{0}$, the negation of all formulas in $\top$ is denoted by $\bot$. We obtain the Boolean algebra $\langle \mathcal{F}^\Gamma / \equiv_\Theta, +, \cdot, -, \bot, \top \rangle$.

To define quantifier algebras two more operations have to be introduced. For $\exists$ the operation $\exists_\kappa$ with $\exists_\kappa(F/\equiv)$ is introduced and denotes the equivalence class of all formulas $(\exists v_\kappa)F$. For quantifier $\forall$ we define a substitution operation S^κ_λ. $S^\kappa_\lambda(F/\equiv)$ denotes the equivalence class of the formula which results from F by replacing each free occurrence of v_κ by v_λ. Extending the Boolean algebra above we obtain the *quantifier algebra of formulas*: $\langle \mathcal{F}^\Gamma / \equiv_\Theta, +, \cdot, -, \bot, \top, S^\kappa_\lambda, \exists_\kappa \rangle_{\kappa,\lambda<\alpha}$ associated with Θ.

If $\mathcal{U}$ is a quantifier algebra as described above a so-called *dimension set* of a formula $a \in A$ is introduced. Quantitatively this is the set $\diamond X$ of all indexes κ of variables v_κ which would change the valuation of formula X if substituted with another variable v_λ. $\mathcal{U}$ is defined to be *locally finite* if $\diamond X$ is a finite set. It can be shown that every quantifier algebra *of formulas* is a locally finite quantifier algebra. Accordingly, one can show that if $\mathcal{U}$ is a locally finite quantifier algebra then there is a theory Θ such that $\mathcal{U}$ is isomorphic to a quantifier algebra of formulas which could be derived from Θ.

This result is already very important as it implies that we can express every theory in FOL without equality by using locally finite quantifier algebras and thus set up a link between algebra and FOL. Conversely, we can take a locally finite quantifier algebra and translate it into a FOL without equality. To emphasise this link we now remove the limitations from above that restricted our first-order formulas to $\mathcal{F}^\Gamma / \equiv_\Theta$.

This can be done by extending the quantifier algebras by another equivalence class, the equivalence class of equality $e_{\kappa\lambda}$. It contains all formulas $v_\kappa \equiv v_\lambda$. This is equivalent to extending Boolean algebra with substitution and existence equivalence classes and we obtain a quantifier algebra with equality: $\langle A, +, \cdot, -, 0, 1, S^\kappa_\lambda, \exists_\kappa, e_{\kappa\lambda} \rangle_{\kappa,\lambda<\alpha}$.

Henkin et al. defined so called *cylindric algebras* in [9]. By changing some of the substitution operators defined above it is relatively easy to show that a cylindric algebra is a quantifier algebra with equality. From [17] we additionally know that axioms defining quantifier algebra with equality are actually all theorems of the theory of cylindric algebras as shown in [9]. Thus, a cylindric algebra is isomorph to a quantifier algebras with equality. Consequentially, it is also possible to map cylindric algebra to FOL with equality. If we only consider FOL without equality Galler also shows in [18] that we can map quantifier algebra to polyadic algebra by slightly modifying the substitution and existence operators.

We showed that we can directly map FOL into different types of algebras and vice versa. Daigneault's interpretation [19] of Krasner's general theory on Galois Fields [20] as polyadic algebras gives even more evidence to the relation of FOL and algebra. This implies again that we can analyse effects in the domains which we can map to algebra in powerful FOL and obtain exciting applications as shown in [21].

C. Temporal Logic

Section III-A quantitatively introduced PL and FOL. Generally speaking these logics support the reasoning based on propositions or terms and formulas. The truth values are fixed and constant over time, i.e. no matter when you evaluate a proposition or a first-order formula, the truth value will always be the same only depending on the valuation function, the propositions, and variables used. Temporal logic extends this classical concept and introduces the dimension of time. Thus, compared with classical logic, which can describe states and properties of systems, temporal logic is also able to express sequences of state changes and properties of behaviour.

Linear Temporal Logic (LTL) extends PL and defines two temporal operators on a set $\mathcal{P}$: the unary operator $\bigcirc$ and the binary operator $Q \cup R$ with $Q, R \in \mathcal{P}$. To give these operators some semantics the regular valuation function from Section III-A is extended by:

1) $v(\bigcirc Q) = 1$, iff in the next time step $v(Q) = 1$.
2) $v(Q \cup R) = 1$, iff $v(Q \wedge \neg R) = 1$ until $v(R) = 1$.

This semantics associates these operators with notions of natural language, namely *until* for $\cup$ and *next* for $\bigcirc$.

Clearly, there are other possible temporal operators that can be derived from these basic operators. This is comparable to the 16 binary operators expressible in PL. If we take a closer look at the definition of these operators we realize that they can be rewritten with "*there exists* a point in time" or "*for all* points in time". Thus, these definitions suggest themselves to ask whether we could model LTL using FOL.

In fact, this is possible but the resulting formulas tend to become very complex and difficult to read. However, the important result to remember is that we can model statements in propositional temporal logic using FOL. Consequentially, we can use the algebras developed in the last section to allow also for the algebraisation of unary-temporal logic.

One obvious question follows: Can formulas in first-order temporal logic be translated into FOL and is it possible to use the same algebras? The answer to this question has been discussed in depth by numerous other contributions. We refer the reader to a good overview and introduction provided in [22] and [23].

LTL gets its name from the fact that it considers only behaviours that can be modelled as linear time sequences. Every state has exactly one successor. However, in in concurrent systems a state in time usually needs to have several future states. To model such systems *branching time logics* (BTL) [24] have been proposed. They possess a tree structure in which each state in time has more than one successor. One of the most popular of these logics is the computation tree logic (CTL) proposed in [25]. These logics often provide specialised path quantifiers which support the "navigation in branches". Nevertheless, sometimes it is easier to use existing tools, proof techniques, and analytical methods that already exist in another domain. Therefore, we can state here that BTL can be translated into linear temporal logic. This implies that branching time logic is a special linear temporal logic and that the same observations which hold for LTL can also be applied to BTLs.

D. Security

We described how temporal logic can extend classical PL to describe time-dependent system characteristics. This expressiveness can be used in many different ways. Very popular is the use of various types of temporal logic in the field of security.

Formal Specification is important in many areas of security research. E.g. distributed computing systems, access control and software systems, security protocols, etc. often use logic to specify security characteristics [26] mainly induced by Pnueli [27], [28] and Lamport [29].

One important characteristic of classical and temporal logic is the existence of a proof calculus which is mainly based on the mathematical foundation of these logics in algebra. This calculus can be used to *formally verify* the correctness of system specifications based on logics.

Specifying the security compliant operation of a system does not only require the thorough specification of its components. It is also required to thoroughly specify how the system is restricted and what the environment can or can not do. Formulas of logic can be used to *describe* these *requirements*.

However, even more important, this work provides links between biology, algebra, and logic. As we intend to use algebra to study structures in biological systems with the special focus on genotypes we can exploit these links to conduct new and innovative research.

We are currently exploring characteristics and structure of and operations on Fraglets [30]. They represent a programming model which can be compared to the copying of DNA sequences. Their execution model is similar to interacting biological systems such as the DNA with the various enzymes that it generates. Due to this similarity, one may expect the implementation of genetic algorithms on these structures to be fairly easy. However, first experiments show that this is not the case [31]. Nevertheless, by choosing a programming model which is very similar to bio-chemical processes in a cell we hope to be able to transfer the observations made in the realm of biology to a programming language, i.e. we try to describe structures of Fraglets using algebra. These structures and their interaction with the environment correspond to specific program characteristics and behaviour. This process is comparable to specific structures of the DNA (genes) which in interaction with their environment yield different phenotypes of an organism. In this way, Fraglets form the exemplary bridge head of the application of our theory to security.

As explained in this section algebra can be mapped into the realm of logic. Depending on the type of algebra we obtain from the analysis of our programming model we will have the possibility to analyse corresponding characteristics using logic. As explained above, logic is a powerful means to investigate program properties, including security. Clearly, this process has its limitations as we will not be able to investigate any arbitrary program and thus security characteristic. However, it will be an important step towards understanding the complicated programming and execution model of Fraglets and possibly their counterpart in biology, the DNA and its proteins. Furthermore, if we invert this analytical process, it becomes clear how we could also guide program evolution. Being able to express specific program properties in logic, including some specific security properties, we will be able to express the same characteristic in algebra and thus as a structure of the programming language.

Consequently, the insights that we obtain from this bridge between biology and logic could also help to improve genetic operators and therewith basically any automated and autonomous code generation process. This is due to the fact that it would become easier to investigate the implications of a structural change (which corresponds to a genetic operation) on the program (security) properties. Evidently, this would also have immediate impact on the design of fitness functions.

IV. Conclusion

The reason we are interested in the metabolism of the cell is that the cell can be considered an immensely complex parallel computer that executes a 'distributed algorithm'. This term arises from the fact that even though most of the instructions are coded in the DNA, a significant part of each metabolic cycle depends on the chemical composition of the cell moment-by-moment. The DNA instructions are propagated through the cell by diffusion mechanisms coupled with various reactions. The concentrations of the various chemical species are far from uniform. In addition, several kinds of membranes and structural elements separate areas of different chemical activity and make the internal topology of the cell nested and extremely complex. Please refer to [10] for a slightly more extensive discussion.

When a gene is activated and begins to signal to the cell machinery to fabricate a particular protein, it creates several thousand mRNA molecules that set an equal number of ribosomes to work in the cytoplasm (each cell has millions of ribosomes, or 'protein factories'). Such a large number of proteins will provide a high probability that the particular function the gene wants to execute will be executed. Therefore, we can regard the large numbers of molecules in the cell as a strategy to achieve a form of dimensional reduction that in computer science we generally call 'abstraction'. Several thousand proteins will participate in a relatively few biochemical reactions to advance one or more metabolic cycles one execution step. Even though the interior of the cell is never in equilibrium (it relies on its 'fall' toward equilibrium as the engine that drives all of its spontaneous self-organising processes—in fact, that's what 'spontaneous' means), its complex topology is divided into many areas in each of which a few reactions are active at any one point in time. From millions of elements we can therefore see how through a relatively small number of quasi-equilibrium regions of the cell several hundred metabolic cycles can be executed in parallel.

Dimensional reduction or abstraction working together with the fact that the DNA itself is composed of genes that can be ON or OFF makes it sound plausible that the internal working of the cell can be modelled through a discrete or digital framework. We can begin to recognise some of the concepts discussed in Section II. For example, as explained in [3] the DNA alphabet of 4 bases can be mapped to finite field $GF(2^2)$. Since each codon of 3 bases can take on 4 values, the set of all possible codons, the DNA code, has 64 elements and can be represented as $GF(2^6)$. Although such a representation may seem arbitrary, by imposing a partial ordering based on the hydrophobicity of the amino acid each codon codes for, Sanchez et al. derive a Boolean lattice analogous to Fig. 2. As verified experimentally, this ordering reflects the robustness of the DNA code with respect to preservation of metabolic function in the presence of mutations. The most common mutations, in fact, result in small steps along this lattice, which correspond to mutant amino acids with similar hydrophobicity. This, in turn, leads to similar protein folding characteristics. As a consequence, the resulting protein is in many cases able to perform the same function.

In conclusion, although we are just at the beginning, the final goal of this research is to be able to specify the security or other functional characteristics of a digital system, and have the specifications map to running code through a process analogous to gene expression guided by universal structural properties of the software 'constituents' in order to achieve spontaneous order construction through interaction with the environment.

V. Acknowledgements

The authors wish to acknowledge valuable feedback from Luciana Pelusi and Marinella Petrocchi on the first draft of this article, and the stimulating conversations with Sotiris Moschoyiannis and Christopher Alm. This research is funded by the EU OPAALS NoE (FP6-034824) and BIONETS IP (FP6-027748).

References

[1] S. Kauffman, *The Origins of Order: Self-Organisation and Selection in Evolution*. Oxford: Oxford University Press, 1993.

[2] P. Dini, *D4.1-DBE Science Vision*. www.digital-ecosystem.org, Deliverables: SP0: DBE Project, 2006.

[3] R. Sanchez, E. Morgado, and R. Grau, "Gene algebra from a genetic code algebraic structure," *Journal of Mathematical Biology*, vol. 51, pp. 431–457, 2005.

[4] P. Dini, *D18.4-Report on self-organisation from a dynamical systems and computer science viewpoint*. www.digital-ecosystem.org, Deliverables: SP5: DBE Project, 2007.

[5] ——, *D18.6-5-Year living roadmap for Digital Ecosystems research in biology-inspired computing*. www.digital-ecosystem.org, Deliverables: SP5: DBE Project, 2007.

[6] G. Boole, *The Mathematical Analysis of Logic*. Bristol: Thoemmes Press, 1998 (1847).

[7] P. R. Halmos, *Algebraic Logic*. AMS, 2007 (1962).

[8] A. Tarski, *Logic, Semantics, Metamathematics*. Oxford University Press, 1956, edited by J. H. Woodger.

[9] L. Henkin, J. D. Monk, and A. Tarski, *Cylindric Algebras*. Amsterdam: North-Holland, 1971.

[10] E. Altman, P. Dini, D. Miorandi, and D. Schreckling, Eds., *D2.1.1: Paradigms and Foundations of BIONETS Research*. BIONETS, 2007. [Online]. Available: www.bionets.org

[11] S. Moschoyiannis, *Group Theory and Error Detecting/Correcting Codes*. Guildford, UK: University of Surrey, Department of Computing, Technical Report SCOMP-TC-02-01, 2001.

[12] P. Cameron, *Introduction to Algebra*. Oxford: Oxford University Press, 1998.

[13] M. Purser, *Introduction to Error Correcting Codes*. Boston: Artech House, 1995.

[14] J. A. Gordon, "Very simple method to find the minimum polynomial of an arbitrary non-zero element of a finite field," *Electronics Letters*, vol. 12, pp. 663–664, 1976.

[15] M. Fitting, *First-order logic and automated theorem proving (2nd ed.)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996.

[16] P. Halmos and S. Givant, *Logic as Algebra*. Washington, DC: The Mathematical Association of America, 1998, vol. 21 of Dolciani Mathematical Expositions.

[17] C. C. Pinter, "A simple algebra of first order logic," *Notre Dame Journal of Fomal Logic*, vol. XIV, no. 3, July 1973.

[18] B. A. Galler, "Cylindric and polyadic algebras," in *Proceedings of the American Mathematical Society*, vol. 8, no. 1, 1957, pp. 176–183.

[19] A. Daigneault, "On automorphisms of polyadic algebras," *Transactions of the American Mathematical Society*, vol. 112, no. 1, pp. 84–130, 1964.

[20] M. Krasner, "Généralisation abstraite de la théorie de galois," in *Algèbre et théorie des nombres*, ser. C.N.R.S. Paris: Centre national de la Recherche scientifique, 1950, no. 24, pp. 163–168.

[21] H. Andréka, I. Németic, and J. Madarász, "On the logical structure of relativity theories." Alfréd Rényi Institute of Mathematics, Budapest, Hungary, Tech. Rep., 2002.

[22] M. Gehrke and Y. Venema, *Algebraic Tools for Modal Logic*. Summer School in Logic, Language and Information, 2001.

[23] P. Blackburn, M. de Rijke, and Y. Venema, *Modal Logic*. Cambridge University Press, 2001.

[24] M. Ben-Ari, Z. Manna, and A. Pnueli, "The temporal logic of branching time," in *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT*. New York, NY, USA: ACM Press, 1981, pp. 164–176.

[25] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach," in *POPL '83*. New York, NY, USA: ACM Press, 1983, pp. 117–126.

[26] F. Kröger, *Temporal logic of programs*. New York, NY, USA: Springer-Verlag New York, Inc., 1987.

[27] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977, pp. 46–57.

[28] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[29] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, May 1994.

[30] C. Tschudin, "Fraglets - a metabolistic execution model for communication protocols," in *2nd Symposium on Autonomous Intelligent Networks and Systems (AINS)*, Menlo Park, USA, July 2003.

[31] L. Yamamoto, D. Schreckling, and T. Meyer, "Self-Replication and Self-Modifying Programs in Fraglets," in *Proceedings of IEEE/ACM Bionetics 2007 (to appear), Budapest, Hungary*, December 2007.