

Hierarchical Code Generators

Mariusz Nowostawski

Information Science Department
University of Otago, Dunedin, New Zealand

Abstract

Hierarchically organised recursive virtual machines can be used as a novel dynamic hierarchies modelling technique. Such a technique is based on the traditional notion of computing machines. The discussed technique allows a detailed analysis of different aspects of hierarchical complex system decomposition, together with the analysis of interactions between and within different hierarchical levels. This may help to understand a modelled problem or phenomenon better. Comparisons with other methods from the field of hierarchical evolutionary computation are provided and simple examples (using Boolean algebra) are presented. An approach to code generation called *software growing* is proposed.

Introduction

In system sciences and cybernetics any system under investigation is thought of as a composition of multiple subsystems, each of which can itself be decomposed into subsystems, and this follows all the way down to a basic, fundamental level (Simon 1968). Hierarchies help us to deal with complex phenomena by decomposing them into more manageable subsystems and investigating the interactions between these subsystems, one interaction at a time. The emphasis is placed on investigation of properties on different levels, mutual dependencies, and interactions between and within the hierarchy levels. Hierarchical decomposition of the problem space deals with complexity in a way that is natural and intuitive to humans.

Many naturally occurring phenomena can be modelled (to an arbitrary precision) by a hierarchy. Building such a hierarchical model is not an easy task, especially if there is a number of interrelated levels involved, and if we do not know all the constraints on the subsystems in advance. Nevertheless, the advantages of hierarchical models, especially for human comprehension, are substantial. For example, human society is composed (in a very simplistic view) of institutions, which are composed of individuals, which are composed of organs, which in turn are composed of cells and tissues, which are further

decomposable into chemical reactions and physical processes. We would not be able to understand various relationships between reactions at molecular levels and reactions at the level of institutions (like company mergers or bankruptcy). Such relationships are implicit and very difficult to deal with directly. They become manageable through hierarchical models. A computer programming is another example of a hierarchical system. Instead of programming computers by means of rearranging the physical wiring between Boolean gates inside digital computers, programmers write software in higher level programming languages, by using compilers/interpreters and virtual machines to logically perform the re-wiring at the (logical) Boolean gates level. These higher-level programs are compiled and executed on lower-level machines, which in turn are executed on specific Boolean gate implementations (hardware). Again, it would be unmanageable to deal with all the higher-level programming abstractions like objects, methods, and attributes in the object-oriented paradigm if it were mapped directly to physical Boolean gates. All the intermediate levels are necessary.

Some believe that all sufficiently complicated systems are modelled best by hierarchical models (Holland 1992; Rosca 1997; Spector 2002). The formalism discussed here aims at providing a uniform framework for modelling dynamic hierarchies.

Virtual machines

The main objective of hierarchy-based models is to analyse and exploit the interactions between different levels. The additional objective for dynamic hierarchy-based models is to construct all necessary hierarchy levels dynamically. This gives extra flexibility.

Formal definitions

The following formalism is based on typical models of computing machines (Papadimitriou 1994; Jones 1997). From the Church-Turing conjecture we know that all models of discrete computation, including the one presented here, have the same properties as any other model

of computation. This fact has some interesting and fascinating implications, see e.g. (Flake 2000). All the well known properties from computational complexity (Li & Vitányi 1997) can be directly applied to the model presented here. This includes, for example, undecidability, the halting problem, and the concept of non-computable functions.

Definition 1. A *virtual machine* or a *computing machine* (or just a *machine* for short) is a tuple $M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s)$ where K is the set of states and $s \in K$ is the initial state. Σ_{in} and Σ_{out} are sets of input and output symbols, respectively, referred to as *input* and *output alphabets*. δ is a function that maps $K \times \Sigma_{in}$ to $K \times \Sigma_{out}$, and is called *the program*. We say δ (or *the program*) *runs on machine* M . Remember that formally δ is an integral part of the machine itself. The notation $M(x)$ represents the output of machine M given the input sequence x . $M(x, y)$ represents the output of machine M given the input sequence x followed by the input sequence y .

Definition 2. Suppose that f is a function from $(\Sigma_{in})^*$ to $(\Sigma_{out})^*$, and let M be a machine with input and output alphabets Σ_{in} and Σ_{out} respectively. We say that M computes f if for any string $x \in (\Sigma_{in})^*$, $M(x) = f(x)$. If such machine M exists, f is called a *recursive function*. We also say that function f is computed by machine M .

Definition 3. If for machine $M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s)$ there exists a machine $M' = (K', \Sigma'_{in}, \Sigma'_{out}, \delta', s')$ which computes δ , we call machine M a *recursive virtual machine* or *recursive machine* for short. We call program δ' an *interpreter* of M , and we say an M interpreter runs on machine M' . We have $\forall x \in (\Sigma_{in})^*$, $M(x) = M'(\delta, x)$.

Definition 4. Suppose we have a machine $M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s)$ and there exists machine $M_c = (K_c, \Sigma_{in_c}, \Sigma_{out_c}, \delta_c, s_c)$, where $\Sigma_{in} \subseteq \Sigma_{in_c}$ and machine $M' = (K', \Sigma'_{in}, \Sigma'_{out}, \delta', s')$ where $\Sigma_{out_c} \subseteq \Sigma'_{in}$ and $\Sigma_{out} \subseteq \Sigma'_{out}$. If $\forall x \in (\Sigma_{in})^*$, $M(x) = M'(M_c(x))$ then we say that δ_c is an M *compiler*, and we say M_c compiles M into M' .

The emphasis in the conceptual framework presented above is to treat algorithms and running programs as *machines* (*recursive virtual machines* to be precise). This along with the notions of compilers and interpreters is discussed in length in (Jones 1997). The above definitions do not make any assumptions about the number of states a given machine can have, nor about the storage capability. All possible models of computations, and different computer/algorithm architectures fit the above definitions. For example one could use $\Sigma \subseteq Real$ to perform analog computation on real values. It can be shown that the proposed conceptual framework is a simple extension of the theoretical models of computation such as Turing machines and

Universal Turing machines (Hopcroft & Ullman 1979; Papadimitriou 1994).

Definition 5. Let machine $M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s)$, with finite input and output alphabets $\Sigma = \Sigma_{in} = \Sigma_{out}$, $\{\sqcup, \triangleright\} \in \Sigma$ and $\{h, y, n\} \in K$. In other words the alphabet contains two special symbols, the blank and the first symbol, and there are three extra state symbols, namely: h the halting state, y the accepting state, n the rejecting state. We define three additional symbols, representing cursor directions: \leftarrow for “left” and \rightarrow for “right” and $-$ for “stay”. If δ maps $K \times \Sigma$ to $K' \times \Sigma$, where $K' = K \times \{\leftarrow, \rightarrow, -\}$ then we say that machine M is a *Turing machine*.

Modelling hierarchies

The modelling methodology proposed in this paper is based on the notion of recursive virtual machines. It is easy to show due to the principle of universality of computation that all virtual machines are recursive. It is also easy to show, again based on the principle of universality, that for any given machine M there exists an interpreter of M and compiler of M to M' . It all collapses at the conceptual level to the universal virtual machine (UVM). Note however, that the universal virtual machine is not the main interest here. Rather it is the concept of machines that are domain-specific, constrained and resource-limited that is the main feature of the model.

We can model artificial and naturally occurring phenomena as a chain of virtual machines. One possible perspective on artificial life or evolutionary systems is to focus on a tower of compilers and/or interpreters. The concepts of chaining and stacking compilers and interpreters is discussed in detail in (Jones 1997). The other approach is to use more traditional functional decomposition. All computing programs, including all evolutionary computation models can be represented as a chain of compilers and/or interpreters, with different functional partitioning on each level. The way this chain is constructed and how all its elements interact with each other is the question we are addressing about hierarchies.

Vertical and horizontal decomposition

Following the formal definitions, a machine can be statically represented as a program string, consisting of a prefix together with some instructions following this prefix. The prefix itself can be decomposed into another prefix and another program, and so on. This is called *vertical decomposition*, or a *vertical hierarchy*. Another type of decomposition is based on dividing a given machine into interacting parts – this is called a *horizontal decomposition*. Formally, a vertical hierarchy is based on stacking interpreters and/or compilers (Jones 1997), see Figure 1.

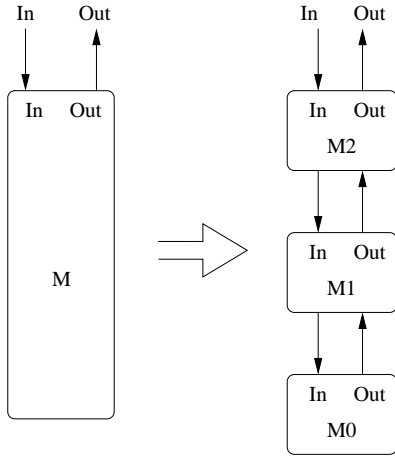


Figure 1: A vertical split of machine M into a tower of machines M0, M1, M2.

A horizontal decomposition is based on splitting a single machine into two or more machines, see Figure 2.

Simple examples of vertical hierarchies are all sorts of (real-life) interpreters and compilers. For example given a Pascal interpreter written in Java we would have: program written in Pascal \rightarrow Pascal virtual machine (written in Java) \rightarrow Java virtual machine (written for example in C) \rightarrow C virtual machine \rightarrow etc., where the arrow reads as “runs on” as defined in Definition 3.

An example of horizontal partitioning would be functional partitioning of a single individual virtual machine. Let us imagine that we have a machine that can compute two operations on the natural numbers domain: addition and multiplication. If we perform functional partitioning, we can end up with two virtual machines, each computing a single operation, multiplication or addition, respectively. The union of these two gives us the original single machine.

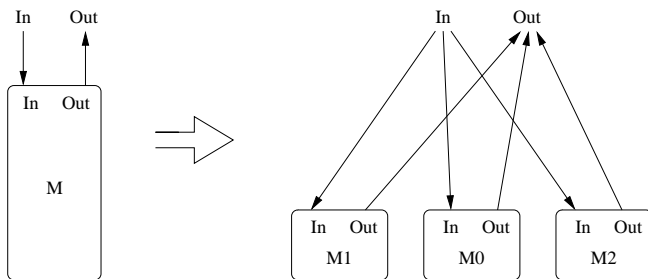


Figure 2: A horizontal split of machine M into machines M0, M1, M2.

One can enumerate through all the machine levels, starting from the base (fundamental) level M_0 , up to the final highest-level machine, M_n . The actual input (instructions) are fed to the machine M_n . It is important

to remember that, in fact, there is no special distinction between the *program* running on a virtual machine and the program emulating a particular machine itself.

All the interacting virtual machines are connected by their input/output streams. The hierarchical structure of that composition can have different forms, depending on the particular phenomena at hand. It can be a simple linear structure or it can be a tree-like structure. In general it is a directed graph, with cycles, with self-referencing nodes, possibly with complicated interdependencies (see Figure 3).

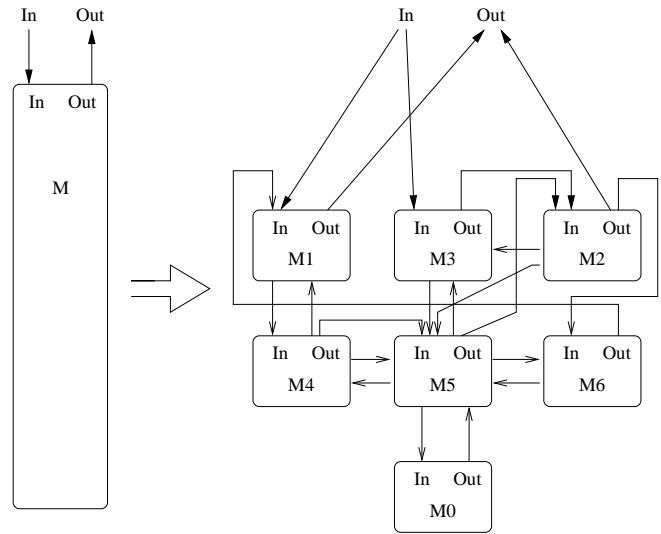


Figure 3: The example of possible dependencies between machines after decomposition

Partial equivalence

Some machines can be fully or partially equivalent to others, for example a Pascal virtual machine written in C and a second one written in Java are always perfect and fully equivalent Pascal virtual machines, even though they use completely different machines on the lower level. Note that even though these two Pascal virtual machines have different machines below them, they can have exactly the same virtual machine one level down, for example a virtual machine for a particular operating system.

One can have a partial Pascal virtual machine that accepts a subset of all possible programs generated in Pascal. This is referred to as *specialisation*. On the other hand it is also possible to have a Pascal virtual machine accepting a subset of expressions from the C language, in addition to normal Pascal programs. The process of adding features to the language and enhancing the input language for a given machine is called *conservative extension* (Jones 1997).

Some machines can be recursively executed on themselves, for example a Java virtual machine interpreter written in Java and executed on a Java virtual machine interpreter. Some machines can be functionally equivalent even though they use completely different language syntaxes or alphabets, for example expressions in prefix, postfix or reverse Polish notations. All these properties are well known in computer science, in which specific languages, interpreters and compilers flourish.

Suppose the problem at hand is coded in such a way that the solution can be expressed as a string of symbols from some language, L . For some languages finding a solution string is easier than for others. Coding of the problem is the key issue in solving the problem. In a sense the language L captures and exploits some of the properties of the problem. This is the main emphasis of the proposed approach. With recursive virtual machines we have the necessary framework to model the transformations of a problem representation from one language to another, and we are able to translate the original problem into a more easily solvable equivalent.

Decomposition limits

A particular level from the hierarchy is treated as a virtual machine that provides some functionality to the other level immediately above it, and uses the level below to have the computation performed. In other words a particular machine accepts input from one level, uses other levels to perform computation, and then returns the results back to yet another adjacent level. The highest level of the chain of machines accepts some input (instructions), interacts with the level below it by sending/receiving some input/output, and returns some outputs (results) back. Similarly to the base level, what we consider the highest level is also arbitrary. There is always a virtual machine feeding the instructions and accepting the results (e.g. a computer program or a human operator).

A given machine in a chain is formally equivalent to an interpreter or compiler of another machine located above it. The first, the base level is the very first interpreter, which we assume as being executed on some universal virtual machine (UVM). In the case of digital computers (and for the sake of simplicity) we can without loss of generality assume that the base level machine is equivalent to the Universal Turing Machine (Hopcroft & Ullman 1979). Of course, this is an arbitrary choice, and the decomposition could be carried further, treating the UVM itself as a virtual machine, running on some software/hardware platform and so on, all the way down to electrical and/or chemical reactions and some physical processes¹.

¹Actually, according to (Fredkin 1992) we have no reason to stop there, and we can decompose the system further,

Evolutionary computation (EC)

Traditional evolutionary program-generation techniques, like artificial life systems or genetic programming with tree-like or linear chromosomes, use human-designed virtual machines with particular properties. The machine is designed to fit into the widest possible class of different problems in a given problem area. Later, a set of “benchmark” problems is tested on a given virtual machine, and inferences are drawn concerning its suitability for a specific class of problems. This methodology works within narrow domains and can be successfully applied to many scientific and engineering problems. Nevertheless, it is stochastic “guesswork”. We propose to unify all existing representations under a single common framework based on the notion of hierarchical virtual machines. This would provide a much more systematic approach and we believe would eventually lead to full automation of the search for an optimal virtual machine hierarchy for a given problem.

Hierarchies in EC

As noted by different authors (Koza 1995; Rosca 1997), all program-generating algorithms use some sort of hierarchical problem decomposition. Otherwise the search space becomes unmanageable due to growing problem complexity. In recent years there has been more interest and work has been done on this particular aspect of managing complexity (Koza 1995; Schmidhuber 1999; Spector 2002).

The basic idea in most cases is to introduce self-referential and recursion manipulation capabilities to the base/fundamental level of the human-designed virtual machine. For example in the PushGP system (Spector & Robinson 2002), the primitive instruction set contains some specific code-manipulation functions that provide recursion and iteration mechanisms. Similar techniques are being used in slightly modified forms in work of other researchers (Schmidhuber 1999; Olsson & Wilcox 2002). The main characteristic of such modelling techniques is that effectively all the levels of abstractions can be collapsed to a single self-referential base level. The *hierarchy levels* are controlled by the initial system settings and they are automatically generated by the mechanisms of the search process. This can improve the search process for some class of problems. However, our objective, which is the analysis of interactions between the hierarchy levels, is not achieved due to the above-mentioned collapse. All the intermediate virtual levels are created automatically and implicitly.

based on the idea that physical phenomena itself are running on some (digital, in the case of Fredkin’s theory) virtual machine.

Evolving recursive virtual machines

The field of evolutionary computation is mainly based on experimentation, a trial and error approach. In light of all the advances in theoretical computer science and given the conceptual framework of recursive virtual machines, it is now possible to introduce a more systematic approach. Each different evolutionary system is an example of a virtual machine, each language is an example of a different search space and each system is an example of the interplay between different aspects of the hierarchical organisation.

Probably one of the closest existing systems using the concept of a tower of virtual machines with a hierarchy of virtual machines is the grammatical evolution system (Ryan, Collins, & O'Neill 1998). In this system, a top-level search is performed on strings of integers. A string containing integers is fed into a particular machine to produce a computer program coded in a particular language as output. This code is then fed as input to yet another machine, which in turn returns a final result. Each of the levels is relative to the level below it; this relativity means that the same top-level string of integers will produce a completely different result when used in combination with another machine. The top-level machine accepting the strings of integers is designed in such a way that it can “plug-in” to any possible second-level machine, and the model will still work. This is a human designed feature, but it is inspired by many naturally occurring phenomena. The multiple levels of indirect influences seems to be the most powerful mechanism at work here.

Instead of designing such machines, and all the indirection levels, by hand, it is believed that this process can be automated, and the virtual machine suitable for a particular class of problems can be discovered automatically. In fact, as shown in the *Example in the Boolean domain* section (below), some of the virtual machines are simple enough that an exhaustive search can be used to discover them.

Seeds and solution growing

Let us take a grammatical evolution system (Ryan, Collins, & O'Neill 1998) as an example of solution growing concept. The solution for a problem at hand is effectively a proper hierarchy of machines (in this case a BNF-encoded language grammar) and a string of integers as a symbolically encoded solution, which we refer to as a *seed*. In case of grammatical evolution system, the hierarchy of machines is designed by human programmer before the search for the proper seed is started. However, the hierarchy of machines needs to be discovered as well, thus it is best to treat it as part of the solution itself.

In general, the solution to the problem (finding a computer program) is a hierarchy of machines together with

the seed. The actual computer program is then generated by feeding the seed through the system. In the case of grammatical evolution, speaking informally, the generation process is (in order): feeding the string of integers, generating the program listing, running the program for the given input and then obtaining the final solution. The given input in this case depends on the “outer-level” virtual machine.

It is, however, possible to change or modify the machine hierarchy just before generating the computer program. If the hierarchy of machines, their connections and the initial states are subject to change, we refer to the process of generating a final solution as *solution growing*. In the case of searching for code one can use the term *code growing* instead. It is possible, by varying the hierarchy of machines, to grow a valid solution from the same seed for a certain variation of the original problem. By simple re-mapping one can achieve exactly the same result by varying the structure of the seed itself. This opens a new window of opportunities not yet used by the automatic code generation techniques. Again, it is a very commonly occurring phenomena in nature.

Formally the idea of *code growing* is based on the notions of *bootstrapping* and *self-application*. This is analogous to more traditional compiler/interpreter bootstrapping and self-application (Jones 1997).

Example in the Boolean domain n-XOR problem

Boolean algebra is widely used for different benchmarks in evolutionary computation (Wong & Leung 1996; Gathercole & Ross 1997; Chellapilla 1998). Within the context of evolutionary computation (Langdon & Poli 1999) present an extended summary of different problems and functions in Boolean algebra. We will investigate one of the more difficult problems: generating a Boolean expression for an *odd-n-parity* problem, referred to here as the n-XOR function. Note, however, that in our set-up for this problem we literally mean n-XOR, where n is the actual input parameter. Traditionally, researchers tackle a particular instance of this problem, for a given n . Our problem is in a sense a meta n-XOR search. The choice of this particular function is based on the observation that this is the most difficult of all 2^n Boolean functions, i.e. the actual expression in conjunctive or disjunctive normal form is the longest. For all other functions the expression is shorter than for n-XOR.

The final solution expression for a given n will be constrained to use only some predefined Boolean operators. n-XOR is a symmetrical Boolean function which returns 1 if and only if the number of activated inputs is odd, and 0 otherwise. The complementary function is called *even-n-parity*. In both cases the main logical operations

to build an n-XOR expression are binary XOR (exclusive OR) and EQ (equality). However to make the problem more difficult, we will constrain the operator set to two 2-variable operations: AND and OR, and one 1-variable operation: NOT. We also require the solution to be of minimal size, as the aim of traditional boolean function minimisation techniques. The size is the length of the expression, in other words, the number of nodes in the expression tree representing the n-XOR function.

Again, note that the prior results discussed below are for a different problem, where only a single instance of the n-XOR truth table is available for fitness evaluation, not the whole class as in our case. The class is usually to be discovered automatically based on a single case example from the class. (Poli & Page 2000) report finding the solutions for fixed n up to 22, however the results are far from being of minimal size. Note, that in their experiments the XOR operator is available, and yet the result for the odd-6-parity problem is almost 5 times longer than the minimal size expression. The author knows of no publication which reports the discovery of a minimal size solution for this problem at all, even with relatively small values of the parameter n , $n > 5$. The best and the closest to the ideal solution is the one informally reported by Roland Olsson. Olsson used his ADATE system (Olsson & Wilcox 2002), and his solution to the same problem (2-variable AND/OR and 1-variable NOT only) for 9-XOR expressions has an impressive 265 nodes, which is only slightly larger than what we believe to be the shortest expression (233 nodes).

Why is that so? Why is the problem “unsolvable” by traditional EC methods? As shown in (Langdon & Poli 1998), the fitness landscape (the search space) for a GP-like system is very rugged and does not allow easy ascent toward optimal solutions. The search space must thus be “transformed” by the GP machine into a more manageable one, which effectively causes code bloat. The code bloat for large n occurs extremely fast, and only methods with proper parsimony pressures can cope with it to a certain extent.

According to (Furst, Saxe, & Sipser 1984), the parity functions cannot be represented by polynomial-size constant-depth AND-OR-NOT expressions, which means that the size of the minimal expression grows with n faster than any polynomial. With 2-arity operations it is also impossible to have a constant-depth solution, which means that the expression depth increases. This makes it extremely difficult for tree search techniques, GP being a perfect example, where nodes closer to the root are very difficult to change. The search space simply becomes unmanageable for larger values of the parameter n .

What can we say about appropriate virtual machines designed to solve the n-XOR problem? We know that

traditional GP-like code generation will simply fail for a large number of variables but can be successfully applied to solve it for small values of n (Langdon & Poli 1998; Poli & Page 2000). The lower bound for the expression size has been proved to be exponential by (Håstad 1989). It is obvious that we have to escape to recursion, otherwise no matter what method we use, the growing search space will become unmanageable for large values of n . We have the interplay of two important aspects: the language in that the expression is expressed and the machine which accepts that language. We also know that our hierarchy of machines has to be dynamic and it has to grow as n grows.

n-XOR hierarchy

Our solution consists of a tower of simple substitution systems. Each machine in the tower accepts the input stream, and for each of the symbols from the input stream it performs a predefined symbol substitution. Each machine can have a different set of rules, or can have exactly the same rules. In the latter case we deal with the same set of rules recursively applied to itself a multiple number of times (self-application). We write the substitution rules as follows: on the left-hand side we write the individual symbol tuple, and on the right-hand side we write the result of the substitution as a tuple. All the inputs and outputs are sequential.

Let us assume that the input stream consists of a sequence of variable names. For a 1-XOR problem we would have a single-variable input stream, say $\langle x_1 \rangle$. For 2-XOR we would have two variables: $\langle x_1, x_2 \rangle$, for 3-XOR three variables: $\langle x_1, x_2, x_3 \rangle$ and so on. In the general case of n-XOR we have $\langle x_1, x_2, \dots, x_n \rangle$. Of course the final expected output is the actual expression representing the n-XOR problem, using only the predefined operators.

In the search for a suitable set of rules, we start with the *divide and conquer* principle, i.e. we try to split our problem into two problems, solve each of them separately, and then combine the results. To do so, first we use an exhaustive search and try to build a shortest solution for the two-variable case. Knowing how to solve the 2-XOR case will allow us to “combine” two sub-solutions later in the process.

Let us estimate first the search space for 2-XOR. We have $t = 2$, where t denotes the number of terminals (variables x_1 and x_2), and $f = 3$, where f donates the number of different operators, which for simplicity we assume to always have two arguments. Thus, according to the formula for the number of different expression trees for a given tree size l (Koza 1992):

$$t^{(l+1)/2} * f^{(l-1)/2} * \frac{(l-1)!}{((l+1)/2)!((l-1)/2)!} \quad (1)$$

we have no more than 55000 different trees of size 9, which yields an estimate on all the trees up to size 9 to be not more than half a million. Our search space for the 2-XOR problem is that big, because the minimal expression for 2-XOR is of length 9. To find a solution we do not need any particular technique, we can use an exhaustive search here, i.e. iterate through all half a million possible solutions.

Notation remark, we use here the following notation conventions: AND and OR are 2-arity Boolean functions, represented by $*$ and $+$ respectively. To simplify the expressions, we omit the explicit representation of the AND operator and assume its operation when two operands are juxtaposed. In equations 2 and 3 all operators are written explicitly to ease counting of the expression size. NOT is a 1-arity operator which is represented as an over-line.

The exhaustive search yields two solutions (assuming the variables are $\langle x_1, x_2 \rangle$):

$$\overline{x_1} * x_2 + x_1 * \overline{x_2} \quad (2)$$

$$\overline{x_1 * x_2 + \overline{x_1 + x_2}} \quad (3)$$

If we rewrite the first of the found solutions as a rule for our substitution system, we will have:

$$\langle x_1, x_2 \rangle \rightarrow \langle \overline{x_1}x_2 + x_1\overline{x_2} \rangle \quad (4)$$

We need also note that the 1-XOR solution is the actual single variable, itself, because the 1-XOR is effectively an identity transformation. We can write that as a rule for halting the substitutions:

$$\langle x_1 \rangle \rightarrow \langle x_1, \perp \rangle \quad (5)$$

With the above two basic rules, found via the exhaustive search, we have built a machine for solving 1-XOR and 2-XOR problems. For 1-XOR it simply halts, presenting the result on the output; for 2-XOR it performs the substitution according to the rule and presents the results to the output.

Now, imagine that we have input that is more than two variables and that our machine can take two input variables that appear sequentially and perform the appropriate substitution to produce an intermediate output which is then passed as input to another machine. Let us assume that all the machines use exactly the same two rules. For the n -XOR problem we would then have a tower of $(n/2) + 1$ such machines.

Let us refer to M_1 as a substitution machine based on the rules above. As an example, if we feed M_1 with two input symbols, the first being $\overline{x_1}x_2 + x_1\overline{x_2}$ and the second being x_3 , we will get as a result:

$$\overline{\overline{x_1}x_2 + x_1\overline{x_2}}x_3 + (\overline{x_1}x_2 + x_1\overline{x_2})\overline{x_3} \quad (6)$$

If N_1 denotes the machine based on the rule from Equation 3, and with the same input strings as above, we will get:

$$\overline{(\overline{x_1}x_2 + x_1\overline{x_2})x_3 + \overline{\overline{x_1}x_2 + x_1\overline{x_2}} + x_3} \quad (7)$$

It takes an exhaustive search to show that there is no better set of rules on each single level than the one we have discovered originally. There is no hidden, or implicit relationship between our dynamically created levels. Because each of the machines in the tower has exactly the same set of rules, we may connect the output of the first machine with its own input, and we will have the case of recursive self-application. The output goes back to the input as feedback. There are effectively two cases now. The first is where there is more than one input string, in which case the first two are read from the input. Then the substituted formula according to the appropriate rule is prepared and the result goes back at the end of the input stream. The second case is if there is only one input string left, in this case this is the solution, and the rule 5 is applied. The effective solution is in a form of a basic recursive function, where the number of recursions depends on the length of the original input. Remember, the original input is the enumeration of all the variables.

The total size of the expression for our substitution system is given by the recursive formula:

$$\begin{aligned} S(1) &= 1 \\ S(n) &= 2S(n\%2) + 2S(n\%2 + n \bmod 2) + 5 \end{aligned} \quad (8)$$

where we have used the notation: $a\%b = \text{floor}(\frac{a}{b})$ and $a \bmod b = a \bmod u\text{lo } b = ((\frac{a}{b}) - (a\%b))b$

Are such simple solutions like the one for n -XOR a common feature for a large class of problems, or is it just one lucky example? The investigation and classification of different problem classes with respect to the hierarchical decomposition seems to be a key to answering this question. From what we have observed so far, it seems to be a common property for all symmetric Boolean functions. The investigation of more problems in Boolean algebra can tell us more about the general properties of this domain.

Summary

A model of dynamic hierarchically organised virtual machines as a modelling technique has been presented. It builds on Turing-machine-based traditional models of computation. The model allows easy transformations from finite-state unlimited-tape machines to infinite-state limited-tape systems, and is not limited to digital computers, by virtue of its unrestricted state space and input/output alphabets. It allows stacking machines (vertical decomposition) in addition to more traditional

functional hierarchical decomposition models. It can be used as a more systematic approach to different code generation techniques. And, as shown by an example, it can easily express recursive properties for some phenomena. Unlike existing models, the emerging levels of organisation can be either modelled directly as individual machines or can be indirectly captured for a formal analysis as a state of an individual machine.

Applications in the fields of evolutionary computation and artificial life are possible and are planned as future work. In particular the formal model presented here allows for the preparation of an operational definition of a living system. However further formalisation of the framework is necessary, and a better understanding of emergent properties is also necessary.

The application of the proposed approach is not limited to these areas though. Some problems are more naturally suited to recursive solutions, however, the hierarchical approach presented above has more general applications than just to those such “naturally recursive” problems. The n-XOR problem is chosen as an example of a “bounded” problem that makes it possible to discuss it fully within the size limitations of this article. However the overall approach is applicable to more general types of problems. It can be effectively used and applied to any problem. In particular, the application of this method to open-ended and continuously recursive processes seems to be promising.

The author would like to thank Stephen Cranefield, Martin Purvis, William B. Langdon, Roland Olsson, and Roy Ward for help, feedback and discussions on different topics related to this article.

References

- Chellapilla, K. 1998. A preliminary investigation into evolving modular programs without subtree crossover. In Koza, J. R.; Banzhaf, W.; Chellapilla, K.; Deb, K.; Dorigo, M.; Fogel, D. B.; Garzon, M. H.; Goldberg, D. E.; Iba, H.; and Riolo, R. L., eds., *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 23–31. University of Wisconsin, Madison, Wisconsin, USA.
- Flake, G. W. 2000. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. MIT Press.
- Fredkin, E. 1992. A new cosmogony: On the origin of the universe. In *PhysComp'92: Proceedings of the Workshop on Physics and Computation*. IEEE Press.
- Furst, M.; Saxe, J. B.; and Sipser, M. 1984. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory* 17(1):13–27.
- Gathercole, C., and Ross, P. 1997. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In Koza, J. R.; Deb, K.; Dorigo, M.; Fogel, D. B.; Garzon, M. H.; Iba, H.; and Riolo, R. L., eds., *Genetic Programming 1997: Proceedings of the Second Annual Conference*, 119–127. Stanford University, CA, USA.
- Håstad, J. 1989. Almost optimal lower bounds for small depth circuits. In Micali, S., ed., *Advances in Computing Research*, volume 5 Randomness and computation. Greenwich: JAI Press Inc. 143–170.
- Holland, J. H. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, reprint edition.
- Hopcroft, J. E., and Ullman, J. D. 1979. *Introduction to automata theory, languages, and computation*. Addison-Wesley Publishing Company, USA.
- Jones, N. D. 1997. *Computability and Complexity: From a Programming Perspective*. MIT Press.
- Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.
- Koza, J. R. 1995. Gene duplication to enable genetic programming to concurrently evolve both the architecture and work-performing steps of a computer program. In Mellish, C. S., ed., *IJCAI*, volume 1. Morgan Kaufmann. 734–740.
- Langdon, W. B., and Poli, R. 1998. Why “building blocks” don’t work on parity problems. Technical Report CSRP-98-17, University of Birmingham, School of Computer Science, Birmingham, UK.
- Langdon, W. B., and Poli, R. 1999. Boolean functions fitness spaces. In Poli, R.; Nordin, P.; Langdon, W. B.; and Fogarty, T. C., eds., *Genetic Programming: Proceedings of EuroGP'99*, volume 1598 of LNCS. Springer Verlag.
- Li, M., and Vitányi, P. 1997. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, New York, second edition.
- Olsson, J. R., and Wilcox, B. 2002. Self-improvement for the ADATE Automatic Programming System. In *Proceedings of the 3rd WSES International Conference on Evolutionary Computation*. WSES Press.
- Papadimitriou, C. H. 1994. *Computational Complexity*. Addison-Wesley Publishing Company, Inc.
- Poli, R., and Page, J. 2000. Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. *Genetic Programming and Evolvable Machines* 1(1/2):37–56.
- Rosca, J. P. 1997. *Hierarchical learning with procedural abstraction mechanisms*. Ph.D. Dissertation, University of Rochester, Rochester, NY 14627, USA.
- Ryan, C.; Collins, J. J.; and O’Neill, M. 1998. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of EuroGP 1998*, 83–96. Springer Verlag.

- Schmidhuber, J. 1999. A general method for incremental self-improvement and multiagent learning. In Yao, X., ed., *Evolutionary Computation: Theory and Applications*. Singapore: Scientific Publishers Co. chapter 3, 81–123.
- Simon, H. A. 1968. *The sciences of the artificial*. MIT Press.
- Spector, L., and Robinson, A. 2002. Genetic programming and autoconstructive evolution with the Push programming language. *Genetic Programming and Evolvable Machines* 3(1):7–40.
- Spector, L. 2002. Hierarchy helps it work that way. *Philosophical Psychology* 15(2):109–117.
- Wong, M. L., and Leung, K. S. 1996. Evolving recursive functions for the even-parity problem using genetic programming. In Angeline, P. J., and Kinner, K. E., eds., *Advances in Genetic Programming 2*. MIT Press. chapter 11, 221–240.