A Systemic Computation Platform for the Modelling and Analysis of Processes with Natural Characteristics

Erwan Le Martelot

Electronic and Electrical Engineering
University College London
Torrington Place, London WC1E 7JE

e.le_martelot@ucl.ac.uk

Peter J. Bentley

Department of Computer Science
University College London
Malet Place, London WC1E 6BT

P.Bentley@cs.ucl.ac.uk

R. Beau Lotto

Institute of Ophthalmology
University College London
11-43 Bath Street, London EC1V 9EL

lotto@ucl.ac.uk

ABSTRACT

Computation in biology and in conventional computer architectures seem to share some features, yet many of their important characteristics are very different. To address this, [1] introduced *systemic computation*, a model of interacting systems with natural characteristics. Following this work, here we introduce the first platform implementing such computation, including programming language, compiler and virtual machine. To investigate their use we then provide an implementation of a genetic algorithm applied to the travelling salesman problem and also explore how SC enables self-adaptation with the minimum of additional code.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development – *Modeling methodologies.*

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence – *Languages and structures*, *Multiagent systems.*

General Terms

Algorithms, Performance, Design, Reliability, Experimentation, Languages, Theory

Keywords

Systemic Computation, Bio-inspired Computation, Genetic Algorithm, Travelling Salesman Problem

1. INTRODUCTION

Does a biological brain compute? Can a real ant colony solve a travelling salesman problem? Does a human immune system do anomaly detection? Can natural evolution optimise chunks of DNA in order to make an organism better suited to its environment?

The intuitive answer to these questions is increasingly: *we think so*. We hold these beliefs through drawing analogies with natural processes and computer algorithms, and demonstrating behaviours and capabilities of the algorithms. Yet what of the processes themselves? No-one has shown that a human brain or an ant colony is Turing Complete. Even if such a proof was developed,

would it be of any use?

Suppose we proved that an ant colony was Turing Complete. This does not solve nor explain the fundamental differences between biological and traditional computation. Just as it is ludicrous to use a real ant colony to model every operation of a modern supercomputer, it is ludicrous for a modern supercomputer to model every operation of an ant colony. The two systems of computation might be mathematically equivalent at a certain level of abstraction, but they are *practically* so dissimilar that they become incompatible.

These differences are important, for natural computation operates according to very important principles. Natural computation is stochastic, asynchronous, parallel, homeostatic, continuous, robust, fault tolerant, autonomous, open-ended, distributed, approximate, embodied, has circular causality, and is complex. The traditional von Neumann architecture is deterministic, synchronous, serial, heterostatic, batch, brittle, fault intolerant, human-reliant, limited, centralised, precise, isolated, uses linear causality and is simple. The incompatibilities are obvious.

To address these issues, [1] introduced *Systemic Computation* (SC), a new model of computation and corresponding computer architecture based on a systemics world-view and supplemented by the incorporation of natural characteristics (listed above). This approach stresses the importance of structure and interaction, supplementing traditional reductionist analysis with the recognition that circular causality, embodiment in environments and emergence of hierarchical organisations all play vital roles in natural systems. Systemic computation makes the following assertions:

- Everything is a *system*
- Systems can be transformed but never destroyed.
- Systems may comprise or share other nested systems.
- Systems *interact*, and interaction between systems may cause transformation of those systems, where the nature of that transformation is determined by a contextual system.
- All systems can potentially act as context and affect the interactions of other systems, and all systems can potentially interact in some context.
- The transformation of systems is constrained by the scope of systems, and systems may have partial membership within the scope of a system.
- Computation is transformation.

Computation has always meant transformation in the past, whether it is the transformation of position of beads on an abacus, or of electrons in a CPU. But this simple definition also allows us

to call the sorting of pebbles on a beach, or the transcription of protein, or the growth of dendrites in the brain, valid forms of computation. Such a definition is important, for it provides a common language for biology and computer science, enabling both to be understood in terms of computation. Previous work [1] has analysed natural evolution, neural networks and artificial immune systems as systemic computation systems and shown that all have the potential to be Turing Complete and thus be fully programmable. In this work we focus on the more applied use of SC, for computer modelling.

SC has been designed to support models and simulations of any kind of nature inspired system, improving the fidelity and clarity of such models. In this paper we introduce the first platform for systemic computation. This platform models a complete systemic computer as a virtual machine within a conventional PC. It also provides an intuitive language for the creation of SC models, together with a compiler. To illustrate the modelling and the mechanism of SC, we present an implementation of a genetic algorithm applied to the travelling salesman problem. We discuss the structure of the systems chosen and the accuracy when various selection and evolution methods are used. We then show how such a SC model can simply be turned into a self-adaptive one.

2. BACKGROUND

Systemic computation is not the only model of computation to emerge from studies of biology. The potential of biology had been discussed in the late 1940s by Von Neumann who dedicated some of his final work to automata and self-replicating machines [2]. Cellular automata have proven themselves to be a valuable approach to emergent, distributed computation [3]. Generalisations such as constrained generating procedures and collision-based computing provide new ways to design and analyse emergent computational phenomena [4][5]. Bio-inspired grammars and algorithms introduced notions of homeostasis (for example in artificial immune systems), fault-tolerance (as seen in embryonic hardware) and parallel stochastic learning, (for example in swarm intelligence and genetic algorithms) [6].

New architectures are also popular, whether distributed computing (or multiprocessing), computer clustering and grid computing and even ubiquitous computing and speckled computing [7]. Thus, computation is increasingly becoming more parallel, decentralised and distributed. However, while hugely complex computational systems will be soon feasible, their organisation and management is still the subject of research. Ubiquitous computing may enable computation anywhere, and bio-inspired models may enable improved capabilities such as reliability and fault-tolerance, but there has been no coherent architecture that combines both technologies. Indeed, these technologies appear incompatible – the computational overhead of most bio-inspired methods is prohibitive for the limited capabilities of ubiquitous devices.

To unify notions of biological computation and electronic computation, [1] introduced SC as a suggestion of necessary features for a computer architecture compatible with current processors, yet designed to provide native support for common characteristics of biological processes. We now present the work achieved in this direction.

3. Overview of Systemic Computation

In systemic computation, everything is a system, and computations arise from interactions between systems. Two systems can interact in the context of a third system. All systems

can potentially act as contexts to determine the effect of interacting systems. One convenient way to represent and define a system is as a binary string. Each string is divided into three parts: two schemata and one kernel. These three parts can be used to hold anything (data, typing, etc.) in binary as shown in Figure 1.
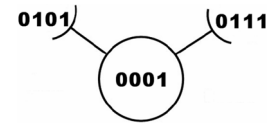
Figure 1. A system used primarily for data storage. The kernel (in the circle) and the two schemata (at the end of the two arms) hold data.

The primary purpose of the kernel is to define an interaction result (and also optionally to hold data). The two schemata define which subject systems may interact in this context as shown in Figure 2.
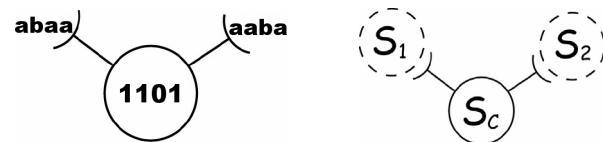
Figure 2. Left: A system acting as a context. Its kernel defines the result of the interaction while its schemata define allowable interacting systems. Right: An interacting context. The contextual system S_c matches two appropriate systems S_1 and S_2 with its schemata and specifies the transformation resulting from their interaction as defined in its kernel.

A system can also contain or be contained by other systems. This enables the notion of scope. Interactions can only occur between systems within the same scope. Therefore any interaction between two systems in the context of a third implies that all three are contained within at least one common super-system.

4. PLATFORM

To realise the SC model a virtual machine (VM) is required. This machine can run any SC program. However, to program the VM, a dedicated programming language and its associated compiler are also necessary. The virtual machine can then run byte-code programs compiled from source code by the compiler. In addition, we introduce a visualisation framework displaying all interaction and computation. This tool aims to provide us with a better understanding of the live "on-line" computation when analysing complex models with intricate components.

4.1 Virtual Machine

A systemic computer (or an SC virtual machine on a conventional computer) runs the "Building Blocks" of Systemic Computation: the systems. Compiled from the program, the systems carry out all computation according to the natural rules of SC.

An SC program differs subtly from conventional logic, procedural or object-oriented program both in its definition and in its goals. A procedural program contains a sequence of instructions to process whereas an SC program needs, by definition, to define and declare a list of agents (the systems), in an initial state. The program execution begins by creating these systems in their initial state and then continues by letting them behave indefinitely and stochastically. The outcome of the program is created from an emergent process rather than a deterministic predefined algorithm.

Since an SC program runs indefinitely, the virtual machine (VM)

has to run an infinite loop. SC is based on parallel, asynchronous and independent systems; therefore the VM can simulate this by randomly picking a context system at each iteration. Once a context is selected, eligible subject systems in the same scope(s) are identified. If any are found, two of them are randomly chosen. A subject is eligible if its definition sufficiently matches the schema of a context. The VM then executes the context interaction instructions to transform the interacting systems and thus process a computation.

Since a context may be isolated from its potential subjects, a computation may not occur at each iteration. It may also not occur if the computing function had nothing to compute from the context and systems it was provided with.

4.2 Language and Compiler

To enable the creation of effective programs for the VM, a language intuitively very close to the SC model has been created together with a compiler translating source code into byte-code for the virtual machine. The aim of the SC language is thus to aid the programmer when defining systems, declaring instances of them and setting scopes between them. The main characteristics of the language are listed below:

● **Defining a system** involves defining its kernel and its two schemata. When a system acts as a context, the two schemata are used as the two templates of the systems to interact with, and the kernel encodes the context behaviour. This raises the problem of coding a schema knowing that it has to specify complete systems (defined by a kernel and two schemata).

The method chosen was to compress information making up each schema [1]. A compression code is used for this purpose, coding three bits (where each bit may be '1', '0' or the wildcard '?') into one character. This allows the complete description of a system (kernel and two schemata) in one single schema, as shown in Figure 2. Here "abaa" and "aaba", in a compression code, describe S1- and S2-like systems respectively.

● **Computing function.** The kernel of each system defines the function(s) to be applied to matching interacting systems (and stores associated parameter values). A lookup table matches binary values with the corresponding binary, mathematical, or procedural call-back functions (defined in the encoding section, see later). These transformation functions are applied to the matching systems in order to transform their values.

● **Labels.** In order to make the code more readable, string labels can be defined and then used in the program instead of their value. These labels must have the correct word length. Since the length bounds the amount of information a system can store, the choice of the length is left to the user.

To combine labels, an OR operator is used in combination with the wildcard symbol. Labels defining different bits within the same schema can then be combined using the '|' operator. For instance if "LABEL_1" and "LABEL_2" are respectively set to "10??" and "??01" then "LABEL_1 | LABEL_2" means "1001". Note that '|' is not a "binary or" since an operation such as "0011 | 1100" is not allowed.

● **Basic system definition.** To illustrate the SC language, Program 1 provides code defining a system similar to the one in Figur. Labels and the "no operation" function NOP, are first defined. The system declaration follows where each line

successively defines the first schemata, the kernel and the second schemata. Any definition is given a name, here "MySystem" to be later referenced when instantiating. In this example, MY_SYSTEM could be a type identifier and MY_K_DATA would be data. NOP represents the nil function making this system unable to behave as a context system. This first system would thus be stored in memory using a string of three words: "0101 0001 0111".

● **Compression operator.** The compression of the template into a coded string does not have to be let to the user. We chose to provide a compression operator '[ ]' to allow the user to write a template like a system definition, without bothering with the compression which will be done automatically at compilation.

● **Full system definition.** Program 3 gives an example of a definition for a system that will behave as context, like the one in Figure 2. In this example, "My_Function" refers to the function to call when two systems interact in the current context. This fragment of program assumes the code of Program 1 is also included. It also assumes the definition of another similar system referred to as "MY_OTHER_SYSTEM". The use of the label "ANY" in the template indicates that here the value of the kernel and the right schemata do not matter in the search for systems to interact with. The other use of the wildcard is shown when combining "My_Function" and "MY_CTX_DATA" using the operator '|'.

Each function defined in a system kernel must refer to an existing call-back function. In the current platform implementation these call-back functions are written in C++ and compiled as plug-ins of the virtual machine. (The only exception is the NOP function.)

● **Encoding section.** The notions of compression code, call-back functions, and word length comprise generic information needed for the compilation and execution of a program. We define them at the beginning of the program file in an "encoding section". As shown in Program 2 the user provides a compression code map file, a list of plug-ins (dynamic libraries ".dll", ".so", etc) containing the transformation functions to be called by the virtual machine, sets the word length and finally sets the offset range used to encode the functions (so that the VM knows where to look in the kernel for the context function reference). In this example, the file "sc_code.map" stores the compression code, the library "my_plugin.dll" owns the call-back functions, the word length is 4 and the function value is in the two first bits of the kernel.

● **System and scope declarations.** Once all the systems have been defined, the last aim of the SC language is to allow the declaration of system instances and their scopes (reminiscent of variable declarations and function scopes in a procedural program). Since scopes are relationships between instances, we propose to handle all this in a "program body". An example, following the previous ones, is given in Program 5.

The first part of the program body declares instances, one by one or in a group (array notation). Note that a system definition name (left part of a declaration) is not a type. An instance (right part of a declaration) is always a system instance initialised with a system definition (triple string value) previously defined and identified by the left name (e.g., MySystem, MyContext). These system values are by definition only initial values which during computation are likely to change. Only inner data such as "MY_ SYSTEM" in Program 1 can be used as a method of typing.

2811

```
label   MY_SYSTEM      0101
label   MY_K_DATA      ??01
label   MY_S_DATA      0111

function     NOP    00??

system MySystem {
      MY_SYSTEM ,
      NOP | MY_K_DATA ,
      MY_S_DATA
}
```

Program 1. Definition of a non-context system.

```
encoding {
      // Characters code map
      char_map "sc_code.map"
      // Functions module(s)
      func_libs { "my_plugin.dll" }
      // 1 character/bit per word
      word_length 4
      // 2 bits for functions (bit 0 to 1)
      func_offset 0:1
}
```

Program 2. An SC program encoding section.

```
label   ANY                   ????
label   MY_CTX_DATA           ??01

function     My_Function   11??

system MyContext {
      [ MY_SYSTEM , ANY, ANY ] ,
      My_Function | MY_CTX_DATA,
      [ MY_OTHER_SYSTEM , ANY, ANY ]
}
```

Program 3. Definition of a context system.

```
system Solution {
      ANY ,
      NOP | SOLUTION ,
      ANY
}

system Operator {
      [ ANY , NOP | SOLUTION , ANY ] ,
      SelectAndEvolve ,
      [ ANY, NOP | SOLUTION , ANY ]
}
```

Program 4. TSP Solution and Operator system declaration.

```
program {

      // Declarations
      Universe       universe ;
      MySystem       ms[1:2] ;
      MyOtherSystem  mos[1:2];
      MyContext      cs[1:2] ;

      // Scopes
      universe { ms[1:2] , mos[1:2] , cs[1:2] }

}
```

Program 5. Systems instantiations and scopes setup.

```
system ComputationSpace {
      ANY ,
      NOP | COMPUTATION_SPACE ,
      ANY
}

system Initialiser {
      [ ANY , NOP | SOLUTION , ANY ] ,
      Initialise ,
      [ ANY , NOP | COMPUTATION_SPACE , ANY ]
}
```

**Program 6. TSP Computation Space and Initialiser definition.**
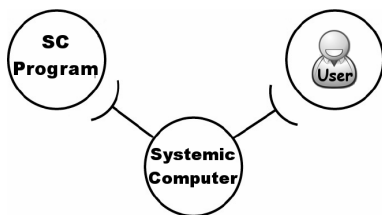
Figure 3. Human-program interaction in the context of the Systemic Computer.

The second part of the program body then sets the scopes between the instances. This notion of scopes refers to embedded hierarchies. An SC program is a list of systems behaving in and belonging to systems which themselves behave in and belong to others and so on. Since the SC definition considers everything as a system, the program is a system, the computer running a program is a system, the user is a system, etc. The human-program interaction can thus be seen as in Figure 3.

A user can interact with a program in the context of a computer. Therefore the program needs to be embedded in a single entity. This leads us to introduce the notion of "universe". Any SC program should have a universe containing everything but itself and being the only one not to be contained. This universe can be defined in any manner, but since it contains everything it cannot

interact by itself with the program. Therefore there is no constraint on its definition and no need for it to act as a context. However, it is the only system a user can interact with. The universe is therefore where user's parameters, to be changed at runtime, should be placed.

Program 5 assumes a system named "Universe" has been defined, although having a dedicated system definition is not mandatory. In this example the universe contains the two instances of MySystem, the two of MyOtherSystem and the two of MyContext, namely everything but itself. This hierarchy is shown in Figure 4.

Dark grey : Universe
White   : MySystem
Black   : MyOtherSystem
Light grey : MyContext
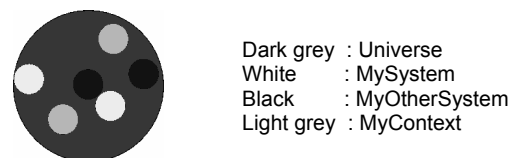
Figure 4. Visualisation of a simple program. The universe encompasses everything.

4.3 Visualisation
Although an SC program does not require any graphical output, understanding embedded hierarchies and computation in massive (hundreds or thousands of) systems can be difficult. Therefore, a

visualisation tool used in parallel with the VM enables a user to gain a better insight of the changing states.

A simple 2D visualisation tool has been created, that represents system hierarchies using a filled circle to represent a system. A visualisation of the program described in the fragments: Program 1, Program 3 and Program 5 can be rendered in 2D as shown in Figure 4.

It is impossible to represent all possible n-dimensional or recursively dimensional (circular hierarchy) environments in a 2D/3D space, however some models can fit or partly fit and thus be well visualised. Such visualisation mainly shows hierarchies and cannot easily render information such as numerical or string values within the spheres. In this case, non-viewable information is provided in a display window as text data, for example Figure 5

Type	Name	Schemata1	Kernel	Schemata2
Universe	universe	????	00??	????
MySystem	ms[1]	0101	0001	0111
MySystem	ms[2]	0101	0001	0111
MyOtherSy...	mos[1]	0111	0001	0111
MyOtherSy...	mos[2]	0111	0001	0111
MyContext	cs[1]	cqzz	1101	dqzz
MyContext	cs[2]	cqzz	1101	dqzz

Figure 5 State (or value) of all systems in Program 5.

The combination of systemic computation model, virtual machine, language and visualiser enables the programmer to create, follow and analyse computation in complex systems.

5. A SYSTEMIC COMPUTATION IMPLEMENTATION OF TSP

The Travelling Salesman Problem (TSP) is a classic problem in the fields of Computational Complexity Theory and Evolutionary Computation (EC). The problem is, given a number of cities and the distance from one city to another, find the shortest round-trip route that visits each city exactly once and then returns to the city of origin.

In EC, genetic algorithms (GAs) or ant colony optimisation (ACO) are commonly used approaches to solve this problem. These are inspired by massively parallel natural processes, yet they are often run on sequential machines. By definition, SC provides us with a stochastic and massively parallel architecture. The aim is thus, using this native feature, to develop a simple and efficient solution to the TSP. We use a genetic algorithm for TSP as a simple test implementation in the SC architecture. The following sections explore the implementation on a systemic computer.

5.1 Systemic Analysis

Systemic computation is an alternative model of computation. Before any new program can be written, it is necessary to perform a systemic analysis in order to identify and interpret appropriate systems and their organisation. When performed carefully, such analysis can itself be revealing about the nature of the problem being tackled and corresponding solution. A systemic analysis is thus the expression of any natural or artificial process in the language of systemic computation.

The first stage is to identify the low-level systems (i.e., determine the level of abstraction to be used). The use of a genetic algorithm implies we need a population of solutions, so a collection of systems, with each system corresponding to one solution, seems appropriate. (A lower-level abstraction might use one system for every gene within each solution, but for the purposes of this investigation, this would add unnecessary complexity.)

The identification of appropriate low-level systems is aided by an analysis of interactions. In a genetic algorithm, solutions interact in two ways: they compete for selection as parents, and once chosen as parents, pairs produce new offspring. The use of contextual systems (which determine the effects of solution interaction) for the genetic operations therefore seems highly appropriate, as shown in Figure 6.
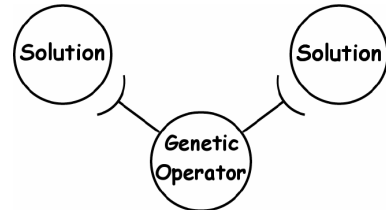

Figure 6. An operator acts as a context for two interacting solutions.
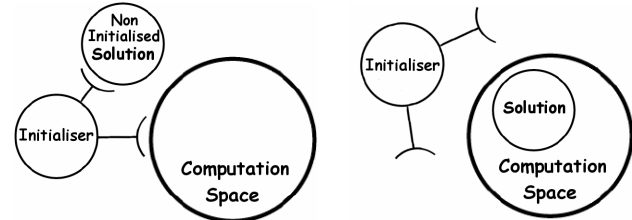

Figure 7. Left: The "Initialiser" acts as context for interactions between non-initialised solutions (located outside the computation space) and the computation space. Right: The result of the interaction, as defined by the initialiser, is an initialised a solution inside the computation space where it can then interact with other solutions in the context of the operators (not shown).

Program 4 provides an example for the definition of solutions and operators. The "SelectAndEvolve" function computes new solutions using the two solutions provided. Any selection and evolution methods may be used. For simplicity, here we perform the selection and reproduction at the same time with the pair of interacting solutions. (All conventional operators could be implemented, e.g. using 'selection' systems to move solutions inside a 'gene pool' system, from which other 'reproduction' systems would control the interaction of parents to make offspring. Such complexity is unnecessary here, however.)

Once the systems and interactions are understood, it is necessary to determine the order and structure of the emergent program. For this we need to determine scopes (which systems are inside which other systems) and the values stored within systems. In a genetic algorithm, the population is usually initialised with random values, before any other kind of interaction can take place. This implies a two-stage computation: first all solutions must be initialised, then they are permitted to interact and evolve. One way to achieve this is to use a 'supersystem' as a computation space, and an initialiser system. If all solutions begin outside the computation space, then the initialiser acts as context for interactions between the empty solutions and the space, resulting in initialised solutions being pushed inside the space ready for evolution, see Figure 7. When declaring the initialiser, the wildcard word ANY is used to fill in the schemata. The computation space and initialiser can be defined as in Program 6.

The systemic analysis is complete when all systems and their organization have been designed for the given problem (or biological process or organism), providing an easy to understand graph-based model. This model can be used on its own as a method of analysis of biological systems, e.g. to understand information flow and transition states. However, a systemic model is best understood when executed as a program. To turn this model into a working program, the data and functions need to be specified for each system in a systemic computation program.

5.2 System Design

5.2.1 Representation and population size.

The representation of the genetic information in a TSP implementation is often chosen as a string of terminals, each of them standing for a city. This is the method of representation we choose here. A chromosome (i.e. solution) is therefore a route (succession of cities). We also use the common method of discarding the initial city. As every city has to be visited once and as all that matters when computing the distance is the order rather than the position, fixing the first city (which therefore does not have to be coded in the chromosome) does not reduce the solution space but reduces the search space and thus the problem complexity.

There are no constraints in SC about how or where data should be stored in a system. In this problem, we store a TSP solution in the left schema, and the distance of the route in the right schema. With the kernel defining NOP, these systems will not act as context for interacting systems matching the schemata (but this would be possible should a different function be defined).

The choice of the number of solutions, or population size, has been addressed in [8], and later specifically for the TSP using a GA in [9]. The aim of the latter was to provide a population size so that the probability that the initial population includes all the edges of an optimum tour is close to one. The population size naturally depends on the problem size. It has also been shown in [9] that the population size in a GA for TSP is not critical, especially if the algorithm includes mutation, and [9] observed that it is not clear what the best population size should be. For our population size we used the estimate provided in [9] (with p = 0.99) which suggests smaller values than Alander's bounds [8].

5.2.2 Genetic operators.

In the computation space, when two solution systems interact in the context of an operator, the operator applies two functions in succession: selection and reproduction.

Because operators exist as entities within the computation space in the same way that solutions do, this provides a novel opportunity for parameter tuning. Since SC randomly picks a context, changing the proportion of operator instances changes the probability of different operator types being chosen. The probability of being chosen, and therefore the impact, of an operator O is thus given by the ratio:

(instances of O) / (total number of operator instances).

In this work, we created several types of genetic operators, which used different approaches for selection, reproduction and mutation. Selection is performed using a simple tournament selection, where the fitness of both solutions is calculated and the winner is the solution with the better fitness. During selection there are two alternatives:

- strictly elitist: the best solution is always kept (KB);

- fitness proportional: the better a solution, the more likely it is to be kept (FP).

It is noticeable that our configuration has many similarities with steady-state GAs. However, the systemic selection chooses two competing solutions at random rather than from a fitness sorted set, which makes it subtly different. (It would be possible to sort and order solutions using 'sort' systems and 'linking' systems to produce an ordered chain of solutions, but again this was deemed unnecessary complexity here.)

After selection, reproduction replaces the less fit solution with a new child, created either by crossover using the two interacting solutions as parents or by duplication and mutation of the selected solution. We chose two different crossover methods:

- partially mapped crossover (PMX) [10],

- ordered crossover (OX) [11].

They were chosen because they are commonly known and usually provide good results. They both guarantee to create only valid chromosomes and to keep the order in the exchanged chunks of genes.

When duplication and mutation is used to generate the child, three alternative methods of mutation were available:

- Swap: chooses two successive cities and swaps them;

- Move: chooses a city and moves it somewhere else;

- Reverse: chooses a route portion and reverses it.

All these operations are commonly used in TSP implementations using GAs. Program 4 defines only one operator as an example. In our implementation we define an operator for each combination of selection and evolution method (i.e. KB or FP using PMX, OX, Swap, Move, Reverse). Therefore, the context function of each operator defines one of the above combinations.

5.3 Experiments and Results

In this section we provide and compare the results of several configurations.

5.3.1 Experiment 1: Selection Comparison

The first experiment we conducted was to assess relative performances of the two selection methods in this implementation. Three identical SC programs were created, one with a KB operator in the computation space, one using an FP operator, and one using both a KB and FP operator. The TSP comprised a well known city map of 48 cities (gr48 from the TSPLIB) for which the best solution is empirically known. Using the formula from [9], a population size of 195 solutions was used, in 1 computation space, with 1 initialiser. The experiment was repeated 10 times, with consistent results. Representative results of one run from this experiment are shown in Figure 8.

As can be seen in Figure 8, while the use of KB enables solutions to converge in a normal manner towards better solutions, the FP method seems to actively prevent evolution, with evolution halting early on. When both operators are used, performance is slightly improved compared to FP, but again evolution halts early.
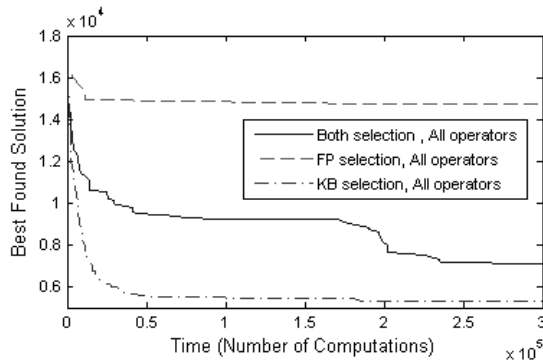
**Figure 8. Comparative test of the two selection methods.**

This result is not surprising if we consider the mechanisms behind the selection methods. Systemic computation by default causes solutions in the same scope to interact randomly in the context of the operator system. Although only two individuals compete at a time, the overall effect on the population is similar to fitness ranked proportional selection. While the KB operator acknowledges this, the FP selection contests it partially. When selecting with FP, if out of the two solutions one is much better than the other, the FP selection will tend to act as a KB selection. If the two solutions are close, the FP selection will be fairly close to random, and randomness is already present when picking the two interacting solutions. It thus appears that the FP operator can only slow down the convergence process without bringing much in return. For this reason, we only use the KB method in further experiments.

5.3.2 Experiment 2: Operator Comparison

In addition to selection, gene diversity depends on the method of reproduction. For example, if operators tend to keep and duplicate the best, the population will quickly tend to contain only copies of this best and may converge prematurely. The aim of the second experiment is thus to evaluate and compare the efficiency of the operators. Using KB selection, six versions of the program were created, each using one of the five operators, and one using all five together (one copy each). Each was executed 10 times with consistent results; Figure 9 provides a plot of one run showing representative results.
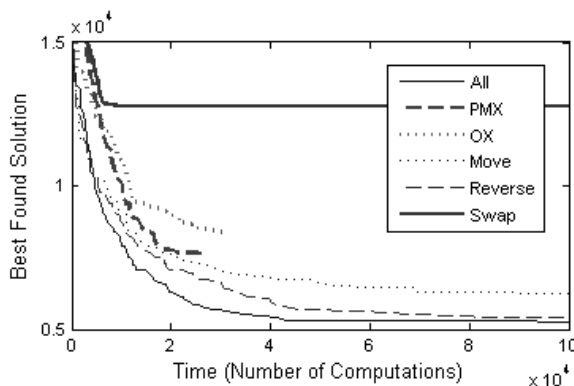
Figure 9. Comparative test of the operators all together and individually.
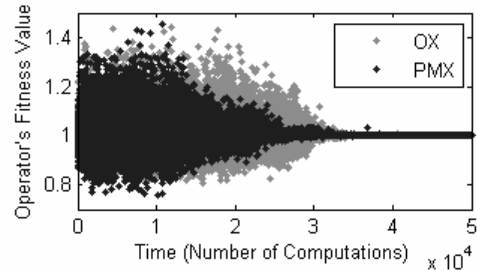
Figure 10. Operators' fitness distribution over time for the crossover operators PMX and OX.

Examining both speed of optimisation, and the ability of the population to keep improving without premature convergence, we can observe in Figure 9 that all operators together perform better in time than any used individually. When used alone, both crossover operators (and especially PMX), perform well but premature convergence seems to occur. Figure 10 shows the effectiveness of OX and PMX over time, by plotting: best parent's fitness / child fitness. Both settle at 1, indicating children with identical fitness to parents, a premature convergence of the population to an imperfect solution.

When analysed, only the reverse mutation seems to enable rapid convergence and continuous improvement thereafter. However, when all operators are available together, the population evolves more effectively.

Indeed, when the contribution of each operator is analysed in more detail (by assessing how frequently each operator produced a fitter solution over time), it becomes apparent that all enable convergence to solutions at different rates, and that those rates all vary over time. Indeed, it can be seen that different combinations of operators would be more appropriate at different times during evolution.

5.4 Self-Adaptive Evolutionary Operators

It is clear that when solutions interact in the right context at the right times, the speed of evolution can be increased, or the ability of evolution to continue making progress can be improved. In nature, factors affecting the progression of evolution (whether part of the evolving organisms, or of the environments of the organisms), are also subject to evolution. The contexts that affect evolutionary progress may co-evolve, giving evolution of evolvability.

If we perform a new systemic analysis and identify the systems, interactions and structure, it is clear that the evolution of evolvability implies new interactions and new systems. Now the genetic operator systems must interact with each other in the context of new operator adapter systems, see Figure 11. This enables the genetic operators to evolve in parallel to the solutions they modify.

In figure 10 we showed how the fitness of individual operators could be assessed by comparing the relative fitness of parent and child solutions produced by an operator. This information is calculated and stored in the operator systems. We thus already have the necessary information to enable the genetic operator adapters to modify the genetic operators.
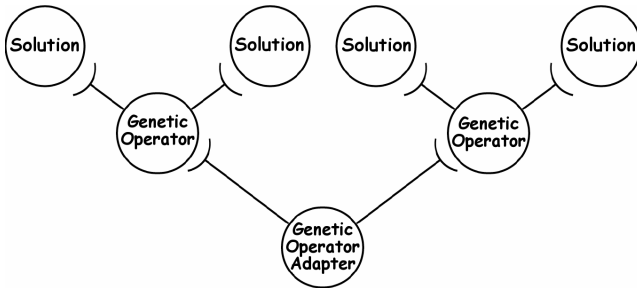
Figure 11. Scheme of an self-adaptive approach to the TSP using a GA on a SC architecture: A Genetic Operator Adapter is added to the current interaction scheme to adapt or evolve Genetic Operators during computation.

We implemented this new feature with a new context system which adapts the operator type between the different approaches. When two operators interact in the context of this system, a fitter operator has on average a higher chance of replacing a less fit one, thus making the number of fitter operators more numerous in the population. The average fitness of a genetic operator is measured within a window of size W last operations. This grows in time following the distribution $W = A.(1-\exp(-k.n))$ where n is the total number of computations performed and A and k are constants set to 1000 and 0.0001 respectively. Lack of diversity was penalised by giving 0.9 instead of 1 when an operator generates an identical copy of the chosen solution.

The system was run using 3 instances of each of the 5 operators, using KB selection, and 1 operator adapter. Figure 12 shows a representative run, comparing the adaptive algorithm with 4 other combinations of operators. It should be clear that the adaptive method consistently outperforms the other approaches; analysis indicates that the relative number of operators changes during the course of evolution.
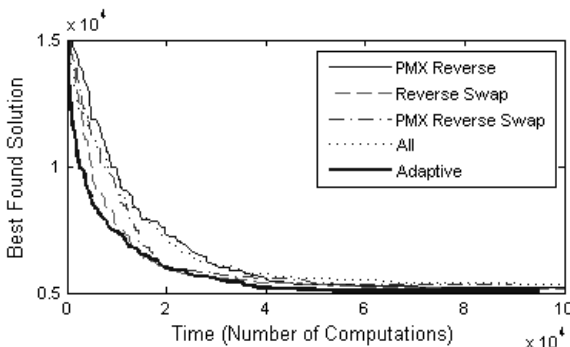
Figure 12. Comparison of the adaptive method with the best tuned methods.

6. CONCLUSION

Biological computation has many significant characteristics that help give it desirable properties. Systemic computation is a model of computation that incorporates those characteristics and suggests a non-von Neumann architecture compatible with conventional hardware and able to support biological characteristics natively.

This paper introduced a platform for the newly introduced systemic computation model including a virtual machine, language, compiler and visualiser. This permits us to design, run and test systems with natural characteristics. This platform was presented together with a concrete application, the travelling salesman problem. We proposed

a genetic algorithm making use of the native characteristics of SC and showed how it could be made self-adaptive with minimal extra code.

Systemic computation is an alternative approach to computation, and can be implemented on any interacting systems, electronic, biological, or mechanical. In addition to being a model of computation, it may also be viewed as a method of analysis for biological systems, enabling information flow, structure and computation within biology to be formulated and understood in a more coherent manner.

Work is still ongoing in this area. It is anticipated that systemic computation may enable a clear formalism of 'complex system.' Another goal is the creation of dedicated parallel hardware for the systemic computer – work is currently underway towards this.

7. REFERENCES

[1] Bentley, P.J . Systemic Computation: A Model of Interacting Systems with Natural Characteristics. In Adamatzky, A., Tueuscher, C. and Asai, T. (Eds) Special issue on Emergent Computation in *Int. J. Parallel, Emergent and Distributed Systems* (IJPEDS), Taylor & Francis pub., Oxon, UK. Vol 22:2.. 2007. pp. 103-121.

[2] J. von Neumann, *The theory of selfreproducing automata*. A. Burks (ed), Univ. of Illinois Press, Urbana 1966

[3] S. Wolfram A New Kind of Science. Wolfram Media, Inc., ISBN 1579550088. (May 14, 2002)

[4] J. H. Holland, Emergence. From Chaos to Order. Oxford University Press, UK. (1998)

[5] A. Adamatzky Computing in Nonlinear Media and Automata Collectives. IoP Publishing, Bristol, 410 pp. ISBN 075030751X. (2001)

[6] G. B. Fogel, D. W. Corne (Eds), Evolutionary Computation in Bioinformatics. Morgan Kaufmann Pub. ISBN: 1558607978 (2003)

[7] D. K. Arvind, K.J. Wong, "Speckled Computing: Disruptive Technology for Networked Information Appliances". In Proc. of the IEEE International Symposium on Consumer Electronics (ISCE'04) (UK), pp 219-223, (September 2004).

[8] Jarmo T. Alander. On optimal populationsize of genetic algorithms. In Patrick Dewilde and Joos Vandewalle, editors, *CompEuro 1992 Proceedings, Computer Systems and SoftwareEngineering, 6th Annual European Computer Conference*, pages 65-70, The Hague, 4.-8. May 1992. IEEE Computer Society Press. 5, 11, 12.

[9] Population Size in GAs for TSP, Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA), 21-23 Aug 1996, Finland

[10] D. E. Goldberg and R. Lingle, "Alleles, loci and the traveling salesman problem", Proceedings of the International Conference on Genetic Algorithms and their Application, Pittsburgh, PA, 1985, pp 154-159

[11] Davis L. "Applying adaptive algorithms to epistatic domains." Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, CA, 1985, pp 162-164