# CRASH-PROOF SYSTEMIC COMPUTING: A DEMONSTRATION OF NATIVE FAULT-TOLERANCE AND SELF-MAINTENANCE

Erwan Le Martelot
Engineering Department
University College London
Torrington Place, London WC1E 7JE
email: e.le_martelot@ucl.ac.uk

Peter J. Bentley
Computer Science Department
University College London
Malet Place, London WC1E 6BT
email: p.bentley@cs.ucl.ac.uk

R. Beau Lotto
Institute of Ophthalmology
University College London
11-43 Bath Street, London EC1V 9EL
email: lotto@ucl.ac.uk

**ABSTRACT**
Reliability in computer or engineering systems is undoubtedly a key requirement in the development process. Safety within critical control systems, and reliable data transfers, require tolerance to unexpected and unwanted phenomena. In biology, new cells can replace damaged cells [1], DNA is able to repair and replicate with error control [1]. These processes are essential to maintain the overall organism. Biology has often been a successful inspiration in computation (artificial neural networks, genetic algorithms, ant colony optimisation, etc) although conventional computation differs widely from natural computation. In this respect, [2] introduced systemic computation (SC), a model of interacting systems with natural characteristics and suggested a new computer architecture. Following this work, [3] introduced a systemic computer as a virtual machine running on conventional computers. In this paper we show, using a genetic algorithm implementation running on this platform, how crash-proof programs following the SC paradigm have native fault-tolerance and easily integrated self-maintenance.

**KEY WORDS**
Distributed and parallel computing and systems, systemic computation, fault-tolerance, self-maintenance, crash-proof, software reliability.

## 1 Introduction

With the increasing performance, potential and complexity in machines and software, it has become increasingly difficult to ensure reliability in systems. Software regularly crashes, top of the line robots break down on the wrong kind of ground, power distribution networks fail under unforeseen circumstances [4]. Yet, there are many approaches to limit potential failures.

In nature, old and potentially damaged cells are constantly being replaced and DNA repaired [1]. The lifespan of cells is shorter than the life of an organism, so fault-tolerance and self-maintenance are essential for the survival of the organism. The failure of some components does not destroy the overall organism; cell death is an important part of staying alive.

To ensure durability, fault tolerance therefore must be as mandatory in a system as its ability to solve a given problem. The latter could actually hardly be trusted or even possible without the former.

Conventional computers are examples of non fault-tolerant systems where the smallest error in code, corruption in memory, or interference with electronics can cause terminal failures [5].

For best reliability, computational systems could mirror biological systems. Recent work at the interface between computer science and biology introduced systemic computation (SC) [2], a new model of computation and corresponding computer architecture based on a systemics world-view and supplemented by the incorporation of natural characteristics. This work was followed by the introduction of a complete platform for this paradigm [3].

In this paper, we show, using a genetic algorithm (GA) implementation on this platform, that SC programs have the native property of fault-tolerance and can be easily modified to become self-maintaining. We compare several variations of the program, involving various faults and self-maintenance configurations, demonstrating that software can repair itself and survive even severe damage.

## 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 [6]. Cellular automata have proven themselves to be a valuable approach to emergent, distributed computation [7]. Generalisations such as constrained generating procedures and collision-based computing provide new ways to design and analyse emergent computational phenomena [8][9]. 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) [2].

New architectures are also popular, whether distributed computing (or multiprocessing), computer clustering and grid computing and even ubiquitous computing and speckled computing [10]. Thus, computation is in-

creasingly 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, [2] 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.

If biology has become very popular in modern computation, fault-tolerant programming is still generally handled in a manner fundamentally different from the methods used in nature.

N-version programming (NVP) [11], or multi-version programming, is a software engineering process that was introduced to incorporate fault-tolerance. Various functionally equivalent programs are generated from the same specifications and compared by the NVP framework. The method thus introduces functional redundancy in order to improve software reliability. However it does not guarantee that the alternative programs are not facing the same issues. It can also make mistakes when, in case of errors, deciding what program version is providing the right answer. Finally, the development and cost overheads are important (several programs for one specification).

Recovery blocks [12] is another technique introduced to provide alternative code blocks to another blocks failing to work properly. It also faces the issues mentioned regarding the previous technique.

With SC, organisms and software programs now share a common definition of computation. We show how this paradigm leads to native fault-tolerance and straightforward self-maintaining programs.

## 3 Overview of Systemic Computation

As introduced in [2], SC is a new model of computation and corresponding computer architecture based on a systemics world-view and supplemented by the incorporation of natural characteristics. 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.

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.

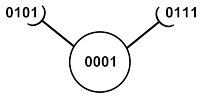


Figure 1. A system used primarily for data storage. The kernel (in the circle) and the two schemata (at the end of the two arms) hold data.

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.

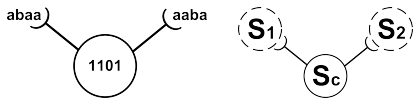


Figure 2. Left: A system acting as a context. Its kernel defines the result of the interaction while its schemata define allowable interacting systems. Right: An interacting context. The contextual system $S_c$ matches two appropriate systems $S_1$ and $S_2$ with its schemata and specifies the transformation 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, where that super-system may be one of the two interacting systems.

## 4 Motivation

SC programming differs subtly from conventional logic, procedural or object-oriented programming both in its definition and in its goals [3]. 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.

Programming with SC has various benefits when regarding fault-tolerance:
- The parallelism of SC means that a failed interaction does not prevent any further interactions from happening,
- A program relies on many independent systems and the failure of one of them cannot destroy the whole program. Like cells in biology, one system collapsing or making mistakes can be compensated by other systems working correctly,
- SC does not permit memory corruption, and even if individual systems contained fatal errors (e.g. divide by zero) the whole program would not halt; every SC program is already in an infinite, never-ending loop so it cannot crash in the conventional sense,
- Having multiple instances of similar systems not only provides redundancy, it also makes it easy to introduce a self-maintenance process which allows similar systems to fix each other, including the self-maintenance systems themselves.

We illustrate these ideas with a concrete program: an implementation of a genetic algorithm (GA) on the SC platform [3]. GAs are a type of bio-inspired search technique used to search for solutions to large or noisy problems. This technique is directly inspired from evolutionary biology involving concepts like genetic inheritance, gene mutation, natural selection and chromosome crossover.

While any program could be used for the demonstration of fault-tolerance and self-repair, we chose a GA as its natural parallelism simplifies the implementation in SC.

## 5 An SC implementation of a GA

When programming with SC it is necessary to perform a systemic analysis in order to identify and interpret appropriate systems and their organisation [3].

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 3.

The operators select the less fit solution and can:
- either replace it with an offspring of the two solutions created by a two-points crossover,
- or apply it a mutation rate of 0.01 bitwise.

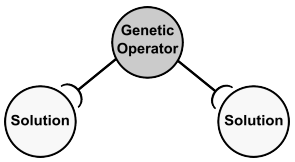


Figure 3. A genetic operator acts as a context for two interacting solutions.

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 super-systems as computation spaces, and initialiser systems. If all solutions begin outside the computation spaces, then the initialiser acts as context for interactions between the solutions outside the spaces, initially empty, and the spaces, resulting in initialising solutions and pushing them inside a space ready for evolution, see Figure 4.

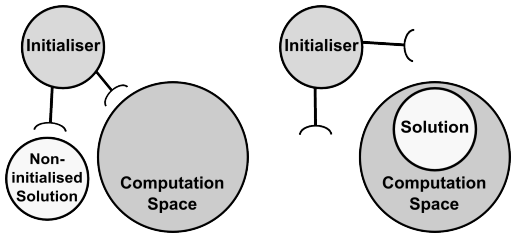


Figure 4. Left: The “Initialiser” acts as context for interactions between non-initialised solutions (located outside any computation space) and a computation space. Right: The result of the interaction, as defined by the Initialiser, is an initialised solution inside a computation space where it can then interact with other solutions in the context of operators (not shown).

One approach in SC modelling, is to implement the input/output layer of a program using a “Universe”, a system which encloses everything within the program. It can only interact with the user through the computer as illustrated in Figure 5. The universe is thus the interface between the program and the user and it is where we will read the output of our GA program.

To transfer data from the program to the user we thus need another system able to provide a relevant output. It can be done by introducing a ”solution transfer” system comparing the solution within a solution system with the solution currently displayed to the user within the universe.

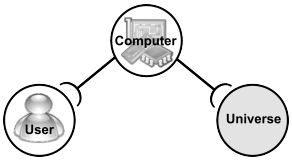


Figure 5. User interaction with the program through the universe in the context of the computer.

If the solution system contains a fitter solution then this solution replaces the one currently being output. To guarantee that the universe and solutions can interact within a solution transfer context, the solution systems as well as the solution-transfers are located within the universe. Therefore, when solutions are pushed in a computation space by the initialisers, they also remain within the universe. Figure 6 shows the global organisation of our GA (with few systems for readability).

Finally, we need to give our GA an aim. In order to observe its progression in a visually convenient way, here the objective is simply to evolve a string of bits that matches a target pattern - a bit string of 256 '1's. (Any other fitness function could have been used instead.)

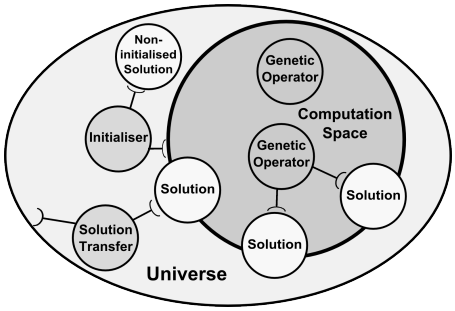


Figure 6. GA program with 1 computation space, 1 initialiser, 1 solution-transfer, 2 operators and 4 solutions (3 initialised and 1 non-initialised). We can see: at the top an initialisation (the solution will then become also part of the computation space); within the computation space two solutions are interacting within the context of a genetic operator; at the bottom left a solution transfer between a solution and the universe. Note that the representation of solution systems between the universe and a computation space means that the solution systems are part of both.

# 6 Fault-tolerance: Experiments & Results

## 6.1 Simulating faults

The aim of this first set of experiments is to study the fault-tolerant behaviour of our program. To achieve this, faults first have to be modelled.

Hardware or software faults can be simulated by randomly altering a memory state (replacing its value with any possible value) with a given rate. By this we provide unpredictable random mistakes that can occur anywhere at anytime in the program.

These faults should be modelled so that their "systemic existence" (i.e. the fact that the systems involved in their modelling exist) does not disrupt the inner organisation of the program. In other words, if we introduce the fault simulation systems in the program with a null fault probability, the program should behave perfectly normally as if the fault systems were absent.

We can state that a fault is due to an unexpected phenomenon which interacts with a component. Whether this phenomenon is an increase of temperature leading to a hardware failure or a programming mistake in memory addressing, the result is the alteration of the memory state in the context of the laws of physics that made this physical change possible. Any program system is therefore susceptible to faults, whether software or hardware initiated.

In SC modelling, the above can be achieved by putting any program system (i.e. system part of our initial program) within a "phenomenon system" also containing the "laws of physics", as shown in Figure 7. The unexpected phenomenon can thus interact with a program system within the context of the laws of physics (the same laws of physics system is within the scope of all phenomena).

In the case of our program, a program system can therefore be a computation space, a solution, an operator, a solution transfer or an initialiser.

The user types in the universe its parameters and read from it what the program returns. This is true for any program. Also, the phenomena and the laws of physics are not part of the "tested program". Therefore we do not consider here the universe, the laws of physics or the phenomena as fallible components.

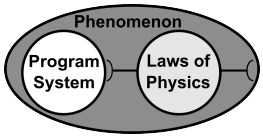


Figure 7. Interaction between an external unexpected phenomenon and a program system in the context of the laws of physics and within the scope of the phenomenon (i.e. the system is encompassed in the field of interaction of the phenomenon).

## 6.2 Experiments

Previous work [5] showed that programs evolved using fractal gene regulatory networks cope better with code damage than human-designed or genetic programming generated programs. In the following experiments, we focus on how human-designed programs for SC can natively cope with code damage.

In the following, iterations, systems, contexts and simple respectively refer to the number of iterations, systems, context systems and non-context systems.

In all experiments, the solutions are 256 bits long. The best solution's fitness is thus 256. To fit solutions within the systems we chose a kernel and schema length

of 256 characters. The total length of a system is thus $ls = 256 \times 3 = 768$.
In the following experiments, all runs for a particular configuration are repeated 10 times, and the presented results are averaged over the 10 runs.
To measure the quantity of errors introduced in the programs we use the following:
- $p_c$: character-wise fault probability,
- $p_s = 1 - (1 - p_c)^{ls}$: system corruption probability,
- $q = p_c.ls.\frac{iterations}{contexts}$: quantity of corrupted bits over an execution.
The ratio $\frac{iterations}{contexts}$ is the number of times each fallible context system can attempt an interaction. It depends on each program configuration and we provide for each experiment an average number that experiments showed to be required for the program to finish.

Also, damages made to context systems have a stronger impact, although as likely to happen as for any other system. Indeed other systems can be scopes and hold no or little data, or data systems usually using less crucial characters than the contexts. It is therefore a useful measure to calculate the "quantities" qc and qs of damage respectively made to fallible context and simple (non context) systems:
- $q_c = q.\frac{contexts}{systems}$: Number of context system bits corrupted in one run,
- $q_s = q.\frac{simple}{systems}$: Number of simple (non-context) system bits corrupted in one run.

**Experiment 1**

Program setup with a minimalist configuration:
- 1 initialiser, - 25 solutions,
- 1 computation space, - 1 crossover operator,
- 1 solution transfer, - 1 mutation operator.

Here $contexts = 4$, $systems = 30$.
10 runs of the program were performed with no fault and 10 runs were performed with faults injected with $p_c = 0.0001$ giving $p_s = 0.0739$
We consider here $\frac{iterations}{contexts} \approx 3700$. Thus, $q(p_c = 0.0001) = 284.16$. We have an estimation of about 284 bits damaged during the program execution, divided amongst the different types of systems as: $q_c = q.\frac{4}{30} \approx 38$ and $q_s = q.\frac{26}{30} \approx 246$.
Figure 8 shows the program progression with and without faults.

**Experiment 2**

The previous program was performing correctly for a very short time due to the single instantiation of all the systems (except the "solution" systems since a GA by definition uses a population of solutions).

The second experiment thus used duplicated systems:
- 5 initialisers, - 25 solutions,
- 3 computation spaces, - 10 crossover operators,
- 10 solution transfers, - 10 mutation operators.

Here $contexts = 35$ and $systems = 63$.
Like in experiment 1, 10 runs of the program were performed with no fault and 10 runs were performed with faults injected with the same probability.
We consider here $\frac{iterations}{contexts} \approx 400$ (many more contexts compared to experiment 1, most of them relevant to the solution computation, i.e. operators). Thus $q(p_c = 0.0001) = 30.72$. We have an estimation of about 31 bits damaged during the program execution divided in: $q_c = q.\frac{35}{63} \approx 17$ and $q_s = q.\frac{28}{63} \approx 14$. Figure 9 shows the results of such configuration tested over 10 runs.

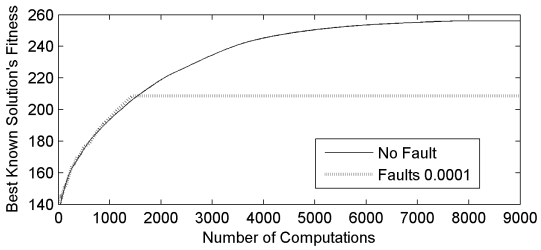

Figure 8. Experiment 1: GA progression without and with faults averaged over 10 runs using a minimalist system configuration. With this first configuration we can see that our GA stops evolving at a very early stage when its program is corrupted by faults. However, it is noteworthy that when injecting faults, the program does not crash; it merely stops evolving properly. The systemic computer is crash-proof as systems deterioration can only stop or corrupt individual interactions, but as the whole program consists of systems interacting in parallel, the other uncorrupted individual interactions will continue as normal.

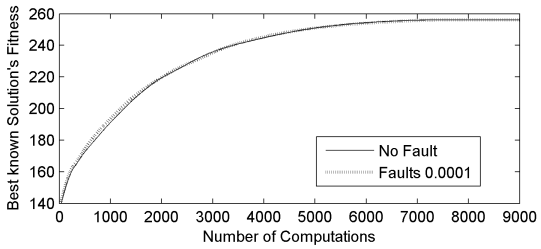

Figure 9. Experiment 2: GA progression without and with faults averaged over 10 runs with a program configuration using redundant systems.

We can now observe that the program performed well in its task in spite of the faults. However, if the execution had required to last longer (e.g. more difficult problem), or if more faults were to occur, the program could stop working before reaching its goal like in experiment 1. This hypothesis is verified in the following experiment.

**Experiment 3**

We use the same systems configuration as in the previous experiment but we rise the character-wise fault probability to $p_c = 0.0005$ giving $p_s = 0.3189$.
This system fault probability is comparable to a simultaneous erroneous state of a third of the computer components.
$q(p_c = 0.0005) = 153.6$
$q_c = q.\frac{35}{63} \approx q.56\% \approx 85 \quad q_s = q.\frac{28}{63} \approx q.44\% \approx 68$

Figure 10 shows the obtained results. We can see that the program, although using duplicated systems, stops evolving before reaching its goal. If we try to analyse the reasons of this program failure we can guess that "solution transfer" systems are the first not to fulfil their role anymore. Initialisers are indeed only required at the beginning, computation spaces are just encompassing systems so have no context nor data holding role, and solutions and operators (crossover or mutation) are more numerous than solution transfers. Analysing the results, looking at the systems memory state evolution through time, showed indeed that each program failure is due in the first place to the corruption of all transfer systems. If solution transfers were more numerous than operators for instance we could then expect solution evolution not to stop working first. As soon as one program subtask (e.g. solution transfer, solution evolution, etc) is not fulfilled any more, the program stops working. In the case of our GA, the subtask in charge of transferring solutions to the universe is not executed anymore once all transfer systems are corrupted. Once such subtask is down, it does not matter what the others can do as the program requires all of them to work properly.

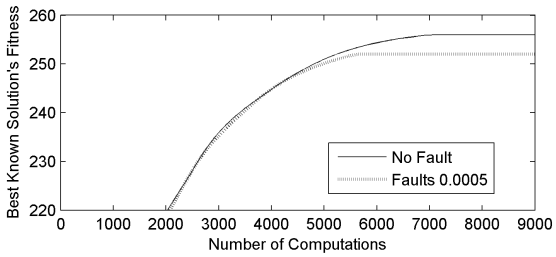


Figure 10. Experiment 3: GA progression without and with faults averaged over 10 runs with a configuration using redundant systems and facing a strong fault probability.

These experiments showed up to now that we always have a graceful degradation (solutions are evolved normally until evolution fails because of damage, but the solutions are not lost) but sooner or later the GA fails to work properly. We can delay the program failure point by providing enough systems to survive for a while (e.g. as long as we need the program to run, see experiment 2) but we cannot prevent this failure if faults keep happening.

To slow down the degradation without adding too many systems (or even avoid this failure point) the program could be repaired. An elegant way to have a program repaired would be an "on-line" self-maintenance of the program. The program would repair itself. No external intervention would then be required as the program would show a homeostatic behaviour.

# 7 Self-Maintenance: Experiments & Results

## 7.1 Implementing self-maintenance

System definitions in our program can be instantiated several times, and need to be in order to provide fault-tolerance. Therefore interacting instances could try to fix each other in a "self-maintenance" context, as shown in Figure 11.

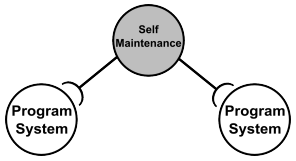


Figure 11. Two program systems interacting within a context of self-maintenance.

Indeed, if the two systems are similar on their healthy parts, then they can replace the damaged parts of each by the ones of the other if these are healthy. The self-repair ability of the program then arises from its conception in independent and multiple times instantiated systems. For this reason, the self-maintenance context systems should also be instantiated several times. The more redundant the information (the more duplicated systems) the more likely systems are to be able to fix each other and the more likely the function they play in the program is reliable.

## 7.2 Experiments

**Experiment 4**

In this experiment, we repeat over 10 runs the same setup as experiment 3 but inject 7 self-maintenance systems. We thus get contexts = 42 and systems = 70.
The amount of self-maintenance systems represents 10% of the total amount of systems
$maintenance = systems.\frac{10}{100} = 7$
We consider here $\frac{iterations}{contexts} \approx 535$, thus $q(p_c = 0.0005) = 205.71$. We have an estimation of about 205 bits damaged during the program execution divided in:
$q_c = q.\frac{42}{70} \approx q.60\% \approx 123 \quad q_s = q.\frac{28}{70} \approx q.40\% \approx 82$
Note that $q_c$ increased with respect to $q$ as this configuration involved additional fallible context systems for self-maintenance. Figure 12 shows the program progression without faults, with faults and then with faults and self-maintenance.

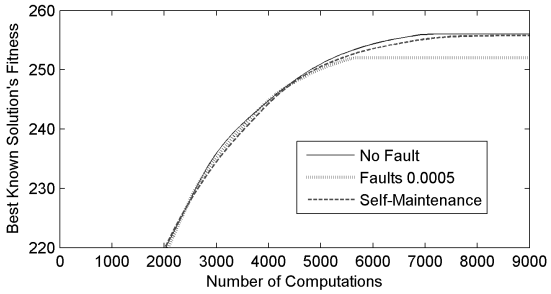


Figure 12. Experiment 4: GA progression with no fault, with faults and with faults and self-maintenance, all averaged over 10 runs with configurations using redundant systems and facing a high fault probability.

We can observe that the program is working fine in

spite of the high amount of faults (e.g. very unreliable hardware or very buggy software), and using a reasonable amount of systems dedicated to faults reparation.

### 7.3 Discussion

It should be noted that the more contexts there are in an SC program, the more iterations are required to make all contexts create an interaction once. Therefore, if the "laws of physics" system interacts once in a cycle, then the more systems there are, the less likely each individual system is to be damaged, but still the probability that something happens within the whole system is the same. We can thus say that faults happen depending on the usage of a system. This bias is part of SC, just as any other paradigm can have inner biases due to their properties.

However, to remove this bias in the experiments, some dummy systems can be added to ensure that experiments with different needs of systems still have the same amount of systems (same amount of context systems and same amount of non-context systems).

To confirm this, experiment 3 was conducted again using 7 dummy context systems in order to have 42 context and 28 non-context systems like in experiment 4. This way the comparison between the two experiments was strictly unbiased. The results showed that the program running with faults performed on average only slightly better than in the dummy-less version, confirming that the bias had no significant impact on the overall outcome of the experiment.

## 8 Conclusion

In this paper, we showed how bio-inspired programming using SC can natively provide fault-tolerant behaviour and easy self-maintenance to a program with minimal software conception overhead: fault tolerance is achieved by duplicating system instances and self maintenance by introducing a new system using existing systems to repair each other. The fault-tolerant self-maintaining GA used in the last experiment showed that we have a crash-resistant computer able to run fault-tolerant self-maintaining programs in spite of a high probability of fault occurrence and allocating only 10% of its resource to maintenance. Therefore, compared to conventional software which crashes immediately, the SC programs are clearly much better.

The overall fault-tolerance is due to the massively distributed architecture and independent computations of the systemic computer making it by definition crash proof, and then to the multiple instantiations of all the systems involved. Finally the self-maintenance systems making use of healthy parts of systems to fix damaged parts of others enabled a homeostatic behaviour.

With this method, fault detection and fault correction are done automatically and is fully integrated into the core of the program. In addition, the fault detection mechanism is independent from the kind of systems being repaired and could therefore be used as it is in any SC program.

Similar to a biological organism, this process is part of the whole and just as any other constituent is a regular and autonomous running task.

In the future, on a hardware systemic computer where systems are physically parallel, software could in addition manage the faults handling process without any real-time overhead.

## References


[1] J.E. Darnell, H.F. Lodish, D. Baltimore, *Molecular Cell Biology*, chap. 5,12 (Scientific Amer Inc, USA, 1990).

[2] P.J. Bentley, Systemic computation, A Model of Interacting Systems with Natural Characteristics, *International journal of parallel, emergent and distributed systems*, 22, 2007, 103-121.

[3] E. Le Martelot, P.J. Bentley, and R.B. Lotto, A Systemic Computation Platform for the Modelling and Analysis of Processes with Natural Characteristics, *Proc of Genetic and Evolutionary Computation Conference (GECCO'07)*, London, UK, 2007, 2809-2816.

[4] P.J. Bentley, Climbing Through Complexity Ceilings, *Invited presentation at Symposium on Distributed Form: Network Practice*, Berkeley, USA., 2004.

[5] P.J. Bentley, Investigations into Graceful Degradation of Evolutionary Developmental Software, *Journal of Natural Computing*, 4, 417-437.

[6] J. von Neumann, *The theory of self-reproducing automata* (Champaign, IL: Univ. of Illinois Press, 1966).

[7] S. Wolfram, *A New Kind of Science* (Champaign, IL: Wolfram Media, Inc., 2002).

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

[9] A. Adamatzky, *Computing in Nonlinear Media and Automata Collectives* (Bristol, UK: Institute of Physics Publishing, 2001).

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

[11] A. Avizienis, The N-Version Approach to Fault-Tolerant Software, *IEEE transactions on software engineering*, SE-11, 1985, 1491-1501.

[12] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, A Program Structure for Error Detection and Recovery, *Operating Systems, Proc of an International Symposium*, 16, Rocquencourt, France, 1974, 171-187.