An artificial immune system approach to automated program verification: Towards a theory of undecidability in biological computing

An immune system inspired Artificial Immune System (AIS) algorithm is presented, and is used for the purposes of automated program verification. Relevant immunological concepts are discussed and the field of AIS is briefly reviewed. It is proposed to use this AIS algorithm for a specific automated program verification task: that of predicting shape of program invariants. It is shown that the algorithm correctly predicts program invariant shape for a variety of benchmarked programs. Program invariants encapsulate the computability of a particular program, e.g. whether it performs a particular function correctly and whether it terminates or not. This work also lays the foundation for applying concepts of theoretical incomputability and undecidability to biological systems like the immune system that perform robust computation to eliminate pathogens.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2690v1 | CC BY 4.0 Open Access | rec: 3 Jan 2017, publ: 3 Jan 2017

An Artificial Immune System Approach to Automated Program Verification: Towards a Theory of Undecidability in Biological Computing

Soumya Banerjee

Abstract — An immune system inspired Artificial Immune System (AIS) algorithm is presented, and is used for the purposes of automated program verification. Relevant immunological concepts are discussed and the field of AIS is briefly reviewed. It is proposed to use this AIS algorithm for a specific automated program verification task: that of predicting shape of program invariants. It is shown that the algorithm correctly predicts program invariant shape for a variety of benchmarked programs. Program invariants encapsulate the computability of a particular program, e.g. whether it performs a particular function correctly and whether it terminates or not. This work also lays the foundation for applying concepts of theoretical incomputability and undecidability to biological systems like the immune system that perform robust computation to eliminate pathogens.

Index Terms — Artificial Immune System, Evolutionary Computing, Program Invariant, Automatic Program Verification, Shape of Invariant, Undecidability, Incomputability, Biological Computing, Immuno-computing

I. INTRODUCTION

The biological immune system has proved to be a rich source of inspiration for computing [1-7, 28-35]. Artificial immune systems take inspiration from the immune system to provide powerful metaphors for robust and distributed computing.

In this paper, I employ an immune system inspired approach to solve a problem in program verification: that of finding a program invariant.

An *invariant* of a program is a mathematical formula that captures the semantics of the program [8] and is used in automatic program verification. The *shape* of an invariant is its approximate polynomial representation. Once the shape of the invariant is predicted, deterministic techniques can be used to generate the exact form of the invariant [9]. Hence, the prediction of invariant shape is of paramount importance for program verification.

An artificial immune system algorithmic framework is proposed to carry out the machine-learning task of predicting invariant shape from an instance of a program. Program invariants encapsulate the computability of a particular program, e.g. whether it performs a particular function correctly and whether it terminates or not. We hope this work will also lay the foundation for applying concepts of theoretical incomputability and undecidability to biological systems like the immune system that perform robust computation to eliminate pathogens [28-35].

II. INTRODUCTION TO CLONAL SELECTION THEORY

A chemical species that can be recognized by the adaptive immune system is known as an antigen (*Ag*). When an organism is exposed to an *Ag*, some specialized immune system cells called B cells respond by producing chemicals called antibodies (*Ab*'s). *Ab*'s are molecules attached primarily to the surface of B cells whose aim is to recognize and bind to *Ag*'s. By binding to these *Ab*'s the *Ag* stimulates the B cell to proliferate and mature into plasma cells that secrete *Ab*. An organism is expected to encounter a given *Ag* repeatedly during its lifetime. The effectiveness of the immune response to secondary encounters is enhanced by the presence of memory cells associated with the first infection, capable of producing high-affinity *Ab*'s after repeat encounters. Such a strategy ensures that the speed and accuracy of the immune response becomes successively higher after each infection. This gives rise to associative memory where the stored pattern is recovered through the presentation of an incomplete version of the pattern. The repertoire of activated B cells is diversified [20]–[23] and B-cells with higher affinity for the antigen are selected to enter the pool of memory cells.

S.B. Author is with University of Oxford, UK (E-mail: soumya.banerjee@maths.ox.ac.uk)

III. AUTOMATED PROGRAM VERIFICATION AND PROGRAM INVARIANTS

The field of automated program verification started with seminal work by Floyd [24] and Hoare [25]. They introduced the concept of a *loop invariant*: a mathematical formula that remains true throughout the execution of a loop. The loop invariant completely captures the semantics of the loop, and along with the program preconditions and postconditions, can be used to show correctness of the program [25].

Previous work [8] has shown how the loop invariant for a particular program can be generated by *a priori* agreement on the *shape of the invariant*: the approximate polynomial representation of the invariant. However, the shape of the loop invariant can be hard to deduce for many programs.

The following shows an example program:

```
{A ≥ 0, B ≥ 0}
x := A;
y := B;
z := 0;

while x > 0 do
  if odd(x) then z := z + y;
  y := 2 * y;
  x := x/2;
end while
```

Assuming the shape of the program invariant as $\mathbf{I_{shape}}$: $Ax + By + Cz + Dxy + Eyz + Fxz + Gxyz + H = 0$, (where A, B, C, D, E, F, G and H are constants or program variables), using quantifier elimination [8] the final loop invariant is $\mathbf{I_{final}}$: $z + xy - AB = 0$. Coupled with a precondition **P**: $\{A \geq 0 \wedge B \geq 0 \wedge x = A \wedge y = B \wedge z = 0\}$ and a postcondition **Q**: $\{z = A*B\}$, it can be shown that this invariant is consistent with **Q** i.e. the program correctly multiplies 2 numbers A and B and stores the result in z.

Finding the precise shape of the loop invariant is generally a non-trivial process and the AIS algorithm proposed aims to use "cues" from the program to make informed predictions about the invariant shape and ultimately help in automated program verification.

IV. PROPOSED COMPUTATIONAL FRAMEWORK

Here we propose a computational framework for predicting program invariants. An AIS algorithm will be used to generate shapes of program invariant. Initially the AIS will be trained on programs, for which the shape of invariant is known. Then a program will be presented to the AIS and it will try to predict the form of the invariant.

An AIS approach presents many advantages over a traditional Machine Learning (ML) approach. In an AIS, recognition can be *sloppy* [26] i.e. if it has previously recognized program P (with an invariant I), then a new program P' "similar" to P, can also be recognized, and an invariant I' can be generated (that is similar in form to I). This is akin to our immune system recognizing a previously encountered pathogen (program), and generating antibodies (invariant) similar to the previously produced antibodies.

The natural immune system produces antibodies by a process of mutation, and the same process is emulated in AIS algorithms. A candidate solution (invariant) will be generated, and then the solution will be improved by *in-silico* mutation.

Previously encountered programs and their corresponding invariants will be stored as memory B cells. When a program similar to a stored one is presented, the time taken to generate the invariant will be shorter than the time taken to generate the original invariant (*secondary response*).

V. COMPONENTS OF THE AIS

Here we define the specific components of the AIS have to be determined. What is the program analogue of an antigen and an antibody?

A *program fragment* is defined to be either an assignment statement, a statement containing an iteration construct (for, while, repeat, etc), or a statement having a conditional check (if <condition> then) e.g. *x := x + 2*, and *while (x > 0) do*, and *if (x > 3) then*, are all program fragments.

The analogue of an antigen is a program fragment and the corresponding analogue of an antibody is an invariant for the program fragment it recognizes. Hence, the AIS will be presented with an antigen (program fragment), and the immune system cells will either produce the antibody (invariant) immediately if it has encountered this antigen before, or will undergo mutations to generate the correct antibody (invariant).

The individual invariants for each program fragment will then be recombined to generate the invariant for the whole program.

VI. A SHAPE SPACE AND ANTIGENIC DISTANCE FOR PROGRAMS

We need a measure of distance between disparate program fragments, so that the AIS can recognize them and generate an antibody in response. For a natural immune system, the antibody combining region relevant to antigen binding can be specified by a number of "shape" parameters [27] which denote the size and shape of the combining site or physical characteristics of the amino acids.

If there are N shape parameters, they can be combined into a vector, and antibody combining sites and antigenic determinants can be described as points **Ab** and **Ag**, in an N-dimensional Euclidean vector-space called *shape space* [27].

Antigenic distance between 2 antigens is the distance in shape space [14] between them e.g. $\|\mathbf{Ag_1} - \mathbf{Ag_2}\|$ is the distance between antigens $\mathbf{Ag_1}$ and $\mathbf{Ag_2}$ in shape space S. The *antibody distance* is the distance $\|\mathbf{Ab_1} - \mathbf{Ab_2}\|$ in shape space between 2 antibodies $\mathbf{Ab_1}$ and $\mathbf{Ab_2}$.

I define the *program fragment shape space* as the N-dimensional Euclidean vector space of program fragment

characteristics like identifier name, exponent on the identifier, operator, etc. I define the corresponding *program fragment antigenic distance* as the distance $||\mathbf{P_1} - \mathbf{P_2}||$ between 2 program fragments $\mathbf{P_1}$ and $\mathbf{P_2}$ in program fragment shape space. The *program fragment antibody (invariant) distance* is the distance $||\mathbf{I_1} - \mathbf{I_2}||$ between 2 program fragments $\mathbf{I_1}$ and $\mathbf{I_2}$ in program fragment shape space.

Let us consider 2 program fragments $\mathbf{P_1}$: $x := x + 2$ and $\mathbf{P_2}$: $t := t + 2$. The corresponding antibody (invariant) for $\mathbf{P_1}$ is $\mathbf{I_1}$: $x = x + 2n$, where n is a program variable or constant (since upon $n - 1$ iterations, x gets the value $x + 2n$). Let $\mathbf{P_1}$ and $\mathbf{I_1}$ constitute the training set. Then the AIS should be able to produce an antibody (invariant) for the program fragment $\mathbf{P_2}$ even though it has never encountered this antigen (program) before. The correct invariant is $\mathbf{I_2}$: $t = t + 2n$ (where n is a program variable or constant) and this is indeed what the AIS generates by somatic hypermutation. The program $\mathbf{P_1}$ differs from $\mathbf{P_2}$ by 1 mutation (replacing x by t on both sides of the assignment) i.e. the *program fragment antigenic distance* $||\mathbf{P_1} - \mathbf{P_2}||$ is 1. The invariants $\mathbf{I_1}$ and $\mathbf{I_2}$ also differ by 1 mutation (replacing x by t) i.e. the *program fragment antibody (invariant) distance* $||\mathbf{I_1} - \mathbf{I_2}||$ is 1. Hence, when an AIS trained on ($\mathbf{P_1}$, $\mathbf{I_1}$) is presented with $\mathbf{P_2}$, it produces $\mathbf{I_2}$ using one mutation from $\mathbf{I_1}$ (Fig. 1).

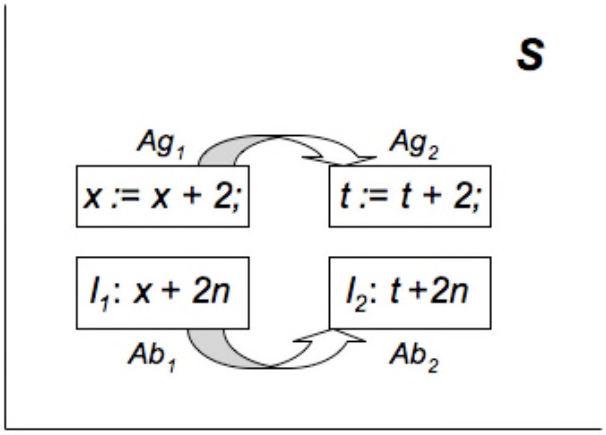

Fig. 1. AIS mutation from the assignment statement $\mathbf{Ag_1}$ (`x := x + 2;`) and invariant $\mathbf{Ab_1}$ (`x + 2n`) to $\mathbf{Ag_2}$ (`t := t + 2;`) and invariant $\mathbf{Ab_2}$ (`t + 2n`) in shape space S.

VII. PROPOSED ALGORITHM

In this section we outline the proposed AIS algorithm. The AIS would be trained on the antigen (program fragment) $\mathbf{P_1}$: $x := x + 2$ and given the antibody (invariant) $\mathbf{I_1}$: $x = x + 2n$ as a solution (*training phase*). The AIS stores the solution $\mathbf{I_1}$ as a memory detector.

When an entire program (as opposed to a program fragment) is presented to the AIS, it breaks the program up into program fragments (all the assignment statements in the program), and then "presents" each of these antigens (fragments) to itself.

If an antigen (program fragment) $\mathbf{P_2}$ "similar" to $\mathbf{P_1}$ is detected, it will generate $\mathbf{I_1}$ as a candidate solution. If $\mathbf{I_1}$ itself does not act as an invariant, the AIS will keep on carrying out randomly on $\mathbf{I_1}$ until it evolves the final antibody (invariant) $\mathbf{I_2}$ that will act as the invariant for the program presented (*somatic hypermutation phase*). This is akin to how the natural immune system mutates B cell receptors and ultimately produces a receptor that can recognize the antigen. The algorithm may also use some heuristics to guide the mutation process e.g. if an antigen (program fragment) of the form `p := p + 5` is encountered, it would search its repertoire for a program fragment that is closest in program shape space to this e.g. `x := x + 5` is closer to the presented antigen (1 mutation) than `y := y + 7` (2 mutations). Additionally, we will have to ensure that each mutation is *sound* i.e. there is no such mutation that would generate a wrong invariant for the corresponding mutated program fragment. In the last step, the AIS incorporates $\mathbf{I_i}$ into its memory pool (*learning phase*).

The AIS then presents the next program fragment $\mathbf{P_3}$, generates the invariant $\mathbf{I_3}$ and stores it in the memory population, and so on until all program fragments have been presented. Finally, the AIS combines all invariants linearly, producing a polynomial (shape of invariant) that captures the semantics of the entire program.

VIII. RESULTS

The AIS (trained on $\mathbf{P_1}$, $\mathbf{I_1}$) presented with suites of entire programs would successfully generate the shape of the invariant. The first program is shown below:

```
(x,y,u,v) := (a,b,b,0);
x := a; y := b;
u := b; v := 0;

while (x ≠ y) do
  while (x > y) do x := x - y; v := v + u;
  end while;
  while (x < y) do y := y - x; u := u + v;
  end while;
end while
```

This program takes 2 positive integers a and b, and calculates their g.c.d and l.c.m. The AIS presents itself with each assignment statement sequentially. The first 4 assignment statements (lines 1-2) have no invariant, since they are not contained inside any loop. Hence, the AIS does not generate any invariant for them. The progress of the algorithm on the next 2 assignment statements (`x := x - y; v := v + u;`) is shown below in Fig. 2.

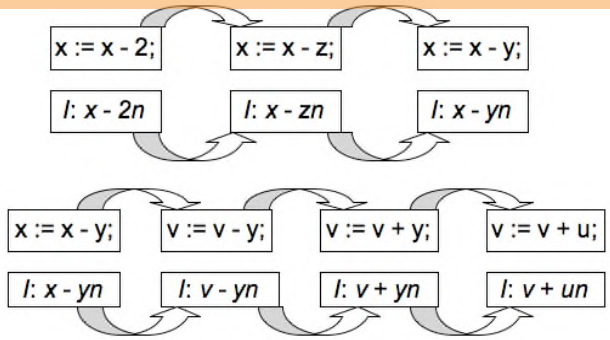

Fig. 2. AIS mutations for the assignment statements `x := x - y; v := v + u;`

The AIS starts from the training set ($\mathbf{P_1}$: $x := x + 2$ & $\mathbf{I_1}$: $x = x + 2n$) and then mutates the operators and operands to create the invariant $\mathbf{I_3}$: $x = x - yn$ for the program fragment $\mathbf{P_3}$: $x := x - y$. The AIS stores $\mathbf{I_3}$ in the memory population and for the next assignment statement (`v := v + u;`), it starts mutating from ($\mathbf{P_3}$,$\mathbf{I_3}$) until it creates the invariant $\mathbf{I_4}$: $v = v + un$ for the program fragment $\mathbf{P_4}$: $v := v + u$.

For the next set of assignment statements (`y := y - x; u := u + v;`), the AIS then generates the invariants $\mathbf{I_5}$: $y = y - xn$ and $\mathbf{I_6}$: $u = u + vn$ (not shown). The 4 invariants $\mathbf{I_1}$, $\mathbf{I_2}$, $\mathbf{I_3}$ & $\mathbf{I_4}$ are then combined linearly (with n being substituted for all program variables, namely x, y, u, v) to yield the invariant shape $\mathbf{I_{shape}}$: $Ax + Bv + Cy + Du + Exy + Fy^2 + Guy + Hvy + Jxu + Ku^2 + Lvu + Mx^2 + Nvx + Pv^2 + Q = 0$, where A, B, C, D, E, F, G, H, J, K, L, M, N, P and Q are constants or program variables. This is the correct invariant shape, since using quantifier elimination [8], the final invariant yielded is $\mathbf{I_{final}}$: $xu + yv - ab = 0$ (with $A = B = C = D = E = F = G = K = L = M = N = P = 0$, $Q = -ab$, $H = J = 1$).

Finally we test the AIS on another standard program [8] shown below:

```
{A ≥ 0, B ≥ 0}
x := A;
y := B;
z := 1;

while y > 0 do
  if odd(y) then y := y - 1;  z := x * z;
  else x := x * x;  y := y/2;
end while
```

This program calculates A^B and stores it in z. The AIS would calculate the invariant for the program fragment $\mathbf{P_5}$: $z := x * z$ as $\mathbf{I_5}$: $z = x^n * z$. For the program fragment $\mathbf{P_6}$: x := x * x, it generates the invariant $\mathbf{I_6}$: $x = exp(x, exp(2,n))$, where $exp()$ is the exponentiation function. Combining all the program fragment invariants, gives us the following invariant shape:

$\mathbf{I_{shape}}$: $Azx^x + Bzx^y + Czx^z + D.exp(x,exp(2,x)) + E.exp(x,exp(2,y)) + F.exp(x,exp(2,z)) + G = 0$.

This is the exact shape of the invariants, since quantifier elimination yields the final invariant

$\mathbf{I_{final}}$: $zx^y = A^B$ (with $A = C = D = E = F = 0$, $G = -A^B$).

We can now readily verify the working of the program. When the loop terminates, the invariant is true and $y = 0$, which yields the correct postcondition: $z = A^B$.

IX. COMPUTATIONAL COMPLEXITY AND OTHER THEORETICAL CONSIDERATIONS

The proposed algorithm would use a sequence of mutations, guided by heuristics, to generate the correct invariant for a program invariant. We can calculate the computational complexity if each mutation is sound and the algorithm for finding an invariant for a program fragment is guaranteed to terminate within n steps, where n is the number of *elements* in a program fragment e.g. the program fragment `y := y + 7` has 3 elements ('y', '+' and '7') and the algorithm will generate the correct invariant in at most 3 mutations in program shape space. Hence if the above assumptions are correct the proposed algorithm will have a computational complexity of O(n) where n is the number of elements in the presented program fragment.

X. CONCLUSION AND FUTURE WORK

We have proposed a computational framework for an immune system inspired approach for automated program verification. The AIS algorithm breaks up a program into fragments and presents them to itself. It then generates an invariant in response to each program fragment and ultimately combines them to create the general shape of the invariant. We show how this approach can be used to generate the general form of the program invariant for non-trivial benchmark programs [8].

Future work will focus on theoretical research into whether there are classes of programs for which a linear combination of individual program fragment invariants might not generate the invariant for the entire program. Another avenue of future investigation would be to look into how mutations on exponentiation would affect the invariant e.g. $x := x + 2$ getting mutated to $x := x^2 + 2$. Lastly, our approach does not consider program fragments having iteration constructs like *while*, *repeat*, etc. and future research will investigate how incorporation of such program fragments can enhance the predictive power of the algorithm.

A lot of work has been done on incomputability, undecidability and program termination in theoretical computer science. The best characterization of this comes in the form of the Halting Problem formulated by Alan Turing. Biological systems also perform computing, e.g. the immune system computes the most efficient way to eliminate pathogens in a timely manner without harming the host [28-35]. However it has been more difficult to define incomputability and undecidability for biological systems.

Program invariants encapsulate the computability and correctness of a particular program, e.g. what it does and whether it terminates or not. This work lays the foundation of applying computability to biological systems especially the

immune system that performs computation. The present work also applies immune system inspired algorithms to find program invariants and prove correctness and termination. In summary, the present work applies the theoretical concepts of undecidability to immuno-computing and possibly biological computing in general.

ACKNOWLEDGMENTS

The author wishes to thank Prof. Deepak Kapur and ThanhVu Nguyen for helpful comments.

REFERENCES

[1] L. N. de Castro and J. Timmis, Artificial Immune Systems: A New Computational Intelligence Approach. London, U.K.: Springer-Verlag, 1996.

[2] J. E. Hunt and D. E. Cooke, "Learning using an artificial immune system," J. Network Comput. Applicat., vol. 19, no. 2, pp. 189–212, Apr. 1996.

[3] D. Dasgupta, Artificial Immune Systems and Their Applications. Berlin, Germany: Springer-Verlag, 1999.

[4] S. A. Hofmeyr and S. Forrest, "Immunity by design: An artificial immune system," in Proc. Genetic and Evolutionary Computation Conf., July 1999, pp. 1289–1296.

[5] L. N. de Castro and F. J. Von Zuben. (1999) Artificial Immune Systems:Part I-Basic Theory and Applications. FEEC/Univ. Campinas,Campinas,Brazil.[Online]. Available:http://www.dca.fee.unicamp.br/~lnunes/immune.html

[6] (2000) Artificial Immune Systems: Part II—A Survey of Applications. FEEC/Univ. Campinas, Campinas, Brazil. [Online]. Available: http://www.dca.fee.unicamp.br/~lnunes/immune.html

[7] de Castro, L. N. & Von Zuben, F. J. (2002), "Learning and Optimization Using the Clonal Selection Principle", IEEE Transactions on Evolutionary Computation, Special Issue on Artificial ImmuneSystems,6(3),pp.239-251.

[8] Kapur, D. Automatically Generating Loop Invariants Using Quantifier Elimination (2004). IMACS Intl. Conf. on Applications of Computer Algebra.

[9] Rodriguez, E, & Kapur, D. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations (2004). International Conference on Symbolic and Algebraic Computation

[10] F. M. Burnet, "Clonal selection and after," in *Theoretical Immunology*,G. I. Bell, A. S. Perelson, and G. H. Pimbley Jr., Eds. New York: Marcel Dekker, 1978, pp. 63–85.

[11] The Clonal Selection Theory of Acquired Immunity. Cambridge,U.K.: Cambridge Univ. Press, 1959.

[12] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction.Cambridge, MA: MIT Press, 1998.

[13] C. A. Janeway, P. Travers, M.Walport, and J. D. Capra, *Immunobiology: The Immune System in Health and Disease*, 4th ed. NewYork: Garland, 1999.

[14] D. J. Smith, S. Forrest, R. R. Hightower, and S. A. Perelson, "Deriving shape space parameters from immunological data," *J. Theor. Biol.*, vol. 189, no. 2, pp. 141–150, Nov. 1997.

[15] P. D. Hodgkin, "Role of cross-reactivity in the development of antibody responses," *Immunologist*, vol. 6, no. 6, pp. 223–226, 1998.

[16] G. L. Ada and G. Nossal, "The clonal selection theory," *Sci. Amer.*, vol. 257, no. 2, pp. 50–57, 1987.

[17] J. Sprent, "T and B memory cells," *Cell*, vol. 76, no. 2, pp. 315–322, Jan. 1994.

[18] D. Mason, "Antigen cross-reactivity: Essential in the function of TCRs"*Immunologist*, vol. 6, no. 6, pp. 220–222, 1998.

[19] S. Haykin, Neural Networks—A Comprehensive Foundation, 2nd sed. Englewood Cliffs, NJ: Prentice-Hall, 1999.

[20] C. Berek and M. Ziegner, "The maturation of the immune response,"*Immun. Today*, vol. 14, no. 8, pp. 400–402, 1993.

[21] A. J. T. George and D. Gray, "Receptor editing during affinity maturation," *Immun. Today*, vol. 20, no. 4, p. 196, 1999.

[22] M. C. Nussenzweig, "Immune receptor editing: Revise and select," *Cell*, vol. 95, no. 7, pp. 875–878, Dec. 1998.

[23] S. Tonegawa, "Somatic generation of antibody diversity," *Nature*, vol. 302, no. 14, pp. 575–581, Apr. 1983.

[24] R. W. Floyd. "Assigning meanings to programs." Proceedings of the American Mathematical Society Symposia on Applied Mathematics. Vol. 19, pp. 19–31. 1967

[25] C. A. R. Hoare. "An axiomatic basis for computer programming". Communications of the ACM, 12(10):576–585, October 1969. doi:10.1145/363235.363259

[26] AS Perelson, FW Wiegel. Some design principles for immune system recognition, Complexity, 1999

[27] A.S. Perelson & G.F. Oster. Theoretical Studies of Clonal Selection: Minimal Antibody Repertoire Size and Reliability of Self-Non-self Discrimination. *J. theor. Biol.* (1979) 81, 645-67

[28] S. Banerjee & M. Moses. Scale Invariance of Immune System Response Rates and Times: Perspectives on Immune System Architecture and Implications for Artificial Immune Systems, *Swarm Intelligence*, 2010

[29] S. Banerjee. Scaling in the Immune System, PhD Thesis, University of New Mexico, USA, 2013

[30] S. Banerjee, D. Levin, M. Moses, F. Koster & S. Forrest, The Value of Inflammatory Signals in Adaptive Immune Responses, *The 10th International Conference on Artificial Immune Systems (ICARIS)*, 2011

[31] Levin D, Forrest S, Banerjee S, Clay C, Cannon J, Moses M & Koster F, A spatial model of the efficiency of T cell search in the influenza-infected lung, *Journal of Theoretical Biology*, 2016

[32] S Banerjee, A Biologically Inspired Model of Distributed Online Communication Supporting Efficient Search and Diffusion of Innovation, *Interdisciplinary Description of Complex Systems*, 2016

[33] S. Banerjee, M. Cebrian, P. van Hentenryck Competitive dynamics between criminals and law enforcement explains the super-linear scaling of crime in cities, *Palgrave Communications*, 2015

[34] S. Banerjee & J. Hecker. A Multi-Agent System Approach to Load-Balancing and Resource Allocation for Distributed Computing. arXiv preprint arXiv:1509.06420, 2015.

[35] S. Banerjee, A Roadmap for a Computational Theory of the Value of Information in Origin of Life Questions, *Interdisciplinary Description of Complex Systems*, 2016

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2690v1 | CC BY 4.0 Open Access | rec: 3 Jan 2017, publ: 3 Jan 2017