Generating Opinion Agent-based Models by Structural Optimisation

A.V. Husselmann

Computer Science, Massey University
Email: a.v.husselmann@massey.ac.nz

***Abstract*—Agent-based modelling has enjoyed a significant increase in research effort in recent years. Particular efforts in the combination of it with optimisation algorithms have allowed the automatic generation of interesting system-level behaviors. The vast majority of these efforts have focussed on parametric optimisation, whereby the structure of a model remains user-defined, and parameters are systematically calibrated. Fairly little effort has been expended in investigations towards combinatorial optimisation in the context of agent-based modelling. The author has previously shown that it is possible to combine the use of a domain-specific language (DSL) and the multi-stage programming paradigm to provide a platform suitable for extension using an evolutionary algorithm. This combination was important to allow run-time code generation, while using just-in-time compiling. The entire process is extremely compute intensive, but can be successfully mitigated using such a language. In this article, three experiments are carried out using this language, and the efficacy of this optimisation is discussed.**

Keywords–Agent-based models; Optimisation; Domain-specific languages; Multi-stage programming.

I. INTRODUCTION

Agent-based modelling (ABM) is significantly multi-disciplinary. It has been used to describe many models which are in essence intuitively reduced to local interactive behavior, which compound over time to generate macro-level phenomena, which are not necessarily specified [1], [2]. Such models famously include Reynolds' "Boids" [3], in which simple interactive rules generate complex behavior reminiscent of flocking birds and schooling fish. Some disciplines which have successfully made use of ABM include medicine [4], [5], political science [6], microbiology [7], and social science more generally [8], [9], as well as extensively across ecology [10], [11], [12].

Models of opinion [13], [14] are particularly well-suited to being studied using ABM. Simple implementations of these models are considered briefly in this article for aiding in optimisation. These include a very simple voter model [15], the Sznajd opinion model [14], and Axelrod's model of cultural dissemination [6]. These familiar models were used in this work as a basis for automatically obtaining new opinion-based models for accomplishing certain objectives given by scalar objective functions.

Domain-specific Languages (DSLs) are making the use of ABM much more streamlined. While no formal definition exists, DSLs are essentially compiled or interpreted languages made specifically for a small target application domain [16]. It is generally agreed that a DSL should be well defined in terms of target domain, syntax, as well as formal, and informal semantics [17]. Indeed, DSLs have already made way into the field of ABM, such as the recent work of Franchi, who presented a DSL built on Python for agent-based social network modelling [18], as well as the NetLogo modelling environment [19]. Typically, ABM modelling packages rely on more general purpose languages such as Java [20], [21].

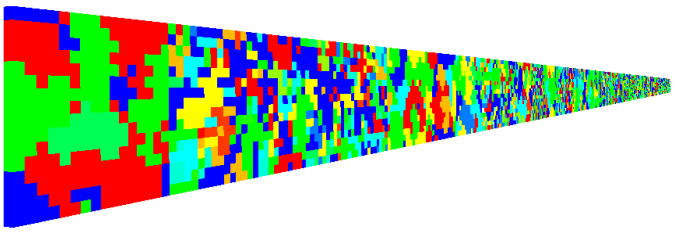

Figure 1. 64, 16x16 side-by-side recombined opinion-based lattice models being evaluated.

An excellent method of creating DSLs while keeping them extensible and fast [22], is the Multi-stage Programming (MSP) paradigm [23]. MSP is particularly attractive for its ability to avoid penalties for run-time code generation (RTCG) [22]. An MSP-based language released recently in 2013 is Terra [24], invented by DeVito et al. It was positioned as the lower-level, high performance counterpart to Lua, a loosely-typed general purpose scripting language [25], [26]. Terra makes use of LuaJIT, which is a just-in-time (JIT) compiler for the Lua language [27]. Terra's implementation of MSP allows the user to write a full compiler architecture in Lua to parse a DSL and generate programs by splicing together Terra code fragments, which are ultimately JIT-compiled before being executed at will.

A DSL (code named MOL) was created using Terra, for the purpose of agent-based model induction, and was presented mid-2014 [28]. This language combines techniques from Gene Expression Programming with the multi-stage paradigm of Terra. It accomplishes this using several stages, involving a parser, type checker, DSL-optimizer, and code generator all written in Lua (and heavily inspired by examples distributed with Terra [24]), with a run-time generator and finally a main Lua program to execute the runtime.

Novelty arises from the combination of this language with a combinatorial optimizer. Genetic Algorithms [29] operate by evaluating a fitness function for a set of candidates in a population [30], and subsequently the population is modified by genetic operators to converge upon an optimal candidate. Similarly here, a population of candidates is evaluated using the runtime, and the main Lua program is then tasked with using a Lua-based optimizer to manipulate candidate programs and recompile them using the DSL compiler stack, and runtime generator. The result of the simulation when compiled with a user interface, is shown in Figure 1. This figure shows multiple opinion-based models being evaluated, where each

 ISBN: 978-1-61208-390-2

model differs from the next in subtle or extreme ways. The best model would be found, and used to generate new models which better suit the objective function provided by the user. The language itself is capable of generating models which are structurally different: in contrast to optimizers which generate models with different parameters.

In the next section, related research and rationale for the approach outlined in this research is presented. In Section III, more detail on the DSL is provided, including its optimisation algorithm. A methodology is then given for evaluating the ability of the language to search through the space of agent-based models for the purpose of inducing some new models. Results of these are given in Section IV. A discussion is provided in Section V. Finally, the article is concluded in Section VI with some areas for future work.

II. Related Research

Previous research attempts to optimise the *structure* of agent-based models using optimisation have not involved DSLs. The recent work of Van Berkel [31], [32] in 2012 involved the use of Grammatical Evolution [33], in order to generate NetLogo [19] programs from predetermined building blocks. While a sophisticated approach, it did not involve a dedicated DSL. A DSL would allow one to prototype different approaches more quickly, rather than re-engineering an existing code base for a different model. Moreover, run-time interpretation of candidate solutions present a significant performance overhead.

Junges and Klügl in 2010, investigated the problem using learning classifier systems, Q-learning and Neural Networks for generating behavior [34]. This was followed in 2011 by their investigation with Genetic Programming [35]. No clear winner among these algorithms was drawn out by the authors, however, they did note in 2012 that Reinforcement Learning and Genetic Programming proved to be more suitable, as they generate human-readable results [36]. It seems appropriate in these circumstances to allow the end-user of such an optimizer a larger breadth of control over the algorithms, whilst still ensuring that it is simple enough to use. This is precisely what Multi-stage Programming allows one to accomplish.

Earlier in 2002, Privošnik developed an evolutionary optimizer which evolved agents with customised finite state machines to solve the Ant Hill problem [37]. While sophisticated, no indication was given concerning reuse of the system for other models. The work of Junges and Klügl however, was integrated with the SeSAm platform to provide a method for use by other researchers [35]. While this is certainly encouraging, the problem of severe performance inadequacies is frequently overlooked. None of these works (except for Van Berkel [31]) extensively consider performance difficulties.

Performance is a very significant issue, and if left unmitigated, can undermine even a sophisticated optimizer. The approach taken in the method described in the next section attempts to solve two problems. One of mitigating excessive computation using parallelism, and the other of still allowing simple interaction with a sophisticated optimizer. The method described is the only known agent-based modelling language with an embedded optimizer, which compiles without alteration to both graphics processing units (GPUs) and single-threaded code without modification. In addition, thanks to Terra and LLVM ("low level virtual machine" compiler architecture) this approach does not suffer run-time interpretation costs, due to generated code being compiled and executed at run-time, for both GPU and CPU. The purpose of this work is to demonstrate and evaluate how well this approach can generate opinion models given a specific objective.

III. Method

The DSL compiler architecture introduced in Section I involves several stages. A flow diagram is provided in Figure 2 which details the process in which special DSL code is compiled. Part of this flow diagram is repeated during run-time, allowing for run-time code generation (RTCG). The process involves three distinct stages. The first is a custom compiler architecture written in Lua, which compiles DSL code to Terra code. Then, using multi-stage programming operators, the Terra code is spliced into a simulation program, and finally compiled using the usual internals of Terra, which involves LLVM. The lattice on which the program operates is updated by this final program exactly once, depending on what update scheme is selected. For example, one may choose between a Monte Carlo style update, or a red-black update style. Clearly these involve different code structures, but fortunately, it is notably easy to accomplish this using the MSP paradigm.

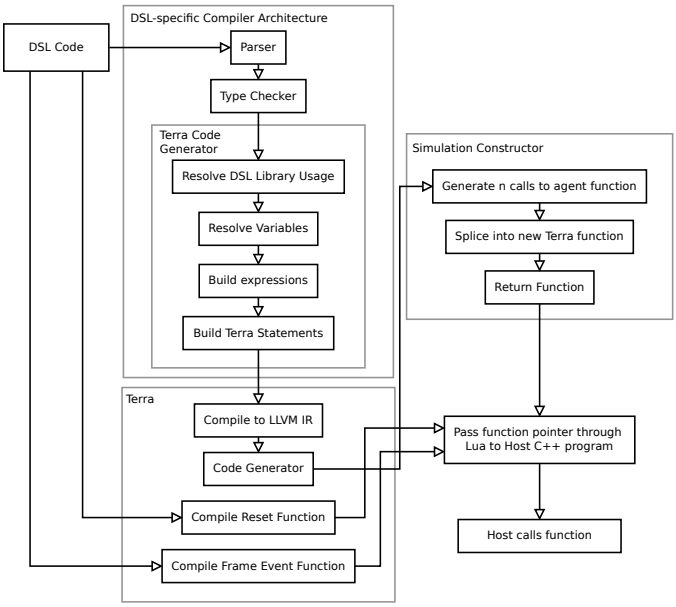

Figure 2. A flow diagram indicating compile-time and some run-time flow.

The final compiled runtime is embedded in a Lua script, which executes the compiled code up to t_n timesteps at a time, before (optionally) rendering the resulting simulations, and at certain points, regenerate all programs by passing the programs through the optimizer, and through the rest of the compiler stack before re-executing the set of programs to obtain new fitness values. The optimizer simply modifies specially marked parts of the Terra expression trees, which are stored in a Lua datastructure.

It has been previously shown that this language and architecture is capable of compiling to NVIDIA PTX code and fully exploit graphics card processing power [28], [38], [39]. The purpose of this article is to cast further light on the

optimisation characteristics of the language, as opposed to its ability to compile to different architectures. Here, the only runtime used is the CPU-based runtime. The complete process is summarised in the algorithm in Figure 3.

```
Allocate large lattice on host
Divide lattice into n rectilinear segments

Parser constructs typed tree from agent prototype code
Type checker checks programs
Platform code generator creates a Terra function f*
Optimiser duplicates f* n times
Simulator code generator creates a stepping function:
for i=0,n do
    Simulator generator inserts call to function i into Terra
    statement list L
end for
L is the simulator Terra code fragment

Runtime generator creates F_r, the runtime Terra function
Inserts L into F_r
Compile F_r (compiles all n Terra trees to machine code)

Begin Lua main program:
Reset fitness scores, reset lattices

for g=0,generation_count do do
    Create Terra program function F_P
    F_P: Insert fragment to zero scores for models
    F_P: Add for-loop for_repeat to F_P for every repeat
    F_P: Insert fragment to reset lattices in for_repeat
    F_P: Insert render call in for_repeat
    F_P: Insert fragment to swap lattices
    F_P: Add for-loop for_timestep to for_repeat
    F_P: Insert call to F_r in for_timestep
    F_P: Insert fragment to swap lattices
    F_P: Insert render call in for_timestep
    Compile the Terra function F_P
    Execute F_P
    Pass programs through optimiser with scores
    Optimiser: Selection (program constructs)
    Optimiser: Recombine selected constructs
    Optimiser: Mutate constructs
    Pass modified programs to simulator code generator
    Pass simulator code to runtime generator
    Recompile the Terra function F_r
    Reset lattices, scores
end for
```

Figure 3. An algorithm describing a complete run of the system for an optimisation test.

As shown in Figure 3, all code generated using the system is generated as fragments of Terra, and spliced together in various stages and various configurations. Once a monolithic Terra function is fully generated, the Terra compiler is used to compile the code at runtime to machine code, using LLVM [40]. When run with the user interface enabled, the compiled program is capable of requesting a re-draw of the simulators. The function is emptied when the user interface is disabled.

The optimisation phase involves a simplified version of the Gene Expression Programming (GEP) algorithm of Ferreira [41], [42]. Whereas the GEP algorithm is designed with several operators for circulating information throughout candidate programs, the recombination optimizer in this system is restricted to a very simplistic minimal set of genetic operators: mutation, crossover and selection. This is purely for convenience at this stage, and will be subject of considerable future improvement.

GEP and the simplified algorithm used here both involve the use of candidates represented as strings of symbols or "codons". It was necessary to be able to translate from a program abstract syntax tree (AST) to a string of codons. The method used to accomplish this is the Karva language, also designed by Ferreira, for the use of GEP [42]. For brevity, this language is not discussed here in great detail. The reader is kindly referred to the excellent book on GEP by Ferreira [42]. Karva expressions, or *k*-expressions, are composed of a head section and tail section. The head-section may contain any symbols, and the tail section may only contain terminal symbols. Terminal symbols are effectively statements which do not involve control flow. Non-terminal symbols correspond to fragments of code which involve control blocks such as *if*-statements. Two additional non-terminals were provided: *P0* and *P1*. These simply execute their arguments sequentially. *P0* is of arity 2, and *P1* is of arity 3.

Three experiments were carried out to cast light on the efficacy of the modelling approach introduced here. Each experiment involved a population of 64 simulations, each one operated on a regular lattice of size 16 by 16. A single evaluation of a model with respect to the objective functions provided was obtained after 50 timesteps, and averaged over 20 (unless otherwise noted) independent runs. The maximum number of evaluation runs (or "generations") was set at 100 during all tests. This number was found to be empirically adequate for obtaining meaningful results, though it would be useful to know in future work how this algorithm behaves with longer runs.

Furthermore, the head length of the *k*-expressions was 3. The update style of the lattices was two-phase with a randomised order, ensuring that all lattice sites executed at least once during a single timestep.

```
select recombination to maximise(score)
 -- pick a new neighbour
 select all
  idx = get_random_neighbour
  s = getneighbour(lattice,idx)
 end

 -- Snzajd Model
 if (me - s) == 0 then
  propagate_n6(newlattice,idx,posx,posy,posz,me)
 end

 -- Axelrod Model
 if randomfloat > (abs(s-me) / OPINIONCOUNT) then
  propagate_single(newlattice,idx,me)
 end

 -- Voter Model
 propagate_single(newlattice,idx,me)
end
```

Figure 4. An excerpt from within the MOL program used in experiments.

 ISBN: 978-1-61208-390-2

Figure 4 contains a fragment of MOL DSL code from within the model code. For brevity, the rest of the model is omitted. Several functions were implemented specially for this model, using an extension framework. Function calls in this code are resolved to functions written in Terra itself in the same scope of the code. get_random_neighbour, however, is a macro, written in Lua, which generates Terra code while having direct access to environment variable references as obtained by the parser.

IV. Experiments

The experiments discussed in the next three sections involve the same code shown in Figure 4, except for the first line. The maximise keyword is used to indicate to the MOL compiler that a score is to be maximised by the optimizer. The expression in brackets, score, is essentially the objective function. The score can be modified at any timestep by any cell agent's program using the special environment variable timestep. The MOL compiler inserts a special statement in the program, which will add the computed score to the current simulation's score. This accumulated score is then later passed to the optimizer for processing.

Each experiment differs in objective function. The first experiment attempts to recombine the Axelrod, Sznajd, and Voter models in order to find a model which minimises the prevailing opinions in the models. That is to say, to find a model with homogeneous agents which converges to a single opinion in 50 time steps. The maximum time steps for this test was set to 50. The second experiment attempts to maximise a cumulative score $s = \sum_{i=0}^{n}(o_i)^{-1}$ where o_i is the number of opinions at timestep i. The third and final model attempts to maximise a more complicated objective function, which involves computing the standard deviation of scores, where each score represents a value indicating the distance from a fully opinion-equalised model.

A. Experiment 1 – Fast Consensus

In this experiment, the objective was to *minimise* the opinion count (ie. obtain consensus) within 50 timesteps. Unlike experiments 2 and 3, this experiment *minimises* the objective function. Here, the objective function is computed as $s(t_n) = o_i$ where o_i is the number of opinions at timestep i, and t_n is the final timestep of the evaluation run. While the compiler will be accumulating all values of $s(t)$ from t_0 to t_n, only $s(t_n)$ will be non-zero for this test. This automatic accumulation is quite useful for quick fitness objectives.

The fitness plot of this experiment is shown in Figure 5. Mean and minimum data are shown. Error bars on the mean indicate the standard deviation of the fitness values in the population. Interesting in this model is the fact that the first generation of programs actually contained a program capable of converging to roughly 4 opinions in 50 timesteps. After approximately 65 generations, a model was generated which could converge to two opinions in 50 timesteps. The results for the run are statistically significant due to the averaging occurring within every evaluation stage. However, it should be noted that this is one sample run of the entire system, and other runs may differ slightly. The purpose in providing sample runs is to qualitatively compare different model structure optimisation approaches particularly by means of different objective functions.

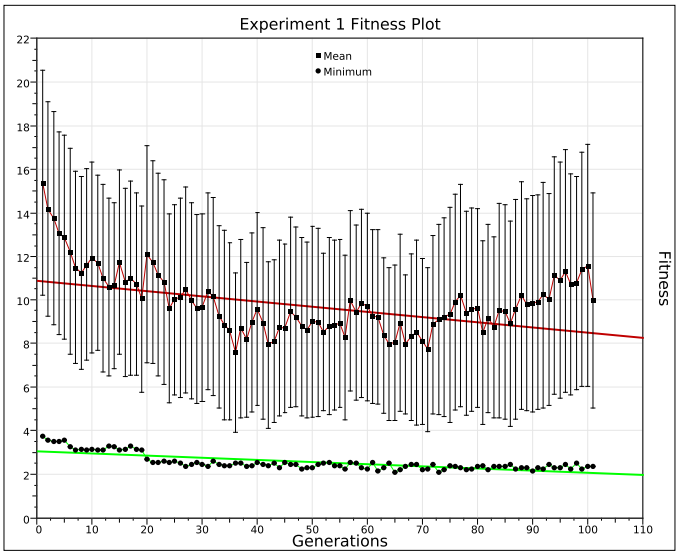

Figure 5. Fitness results by generation for experiment 1.

As seen in Figure 5, search stagnation prevents the algorithm from generating a model which converges to a single opinion. This could also be due to the specification of the model code in the optimisation structure. This is a significant disadvantage of the MOL language. It is possible that a user may provide insufficient information in the structure, and therefore the optimizer can never reach an optimal result. It is also possible that 50 timesteps is simply insufficient for reaching consensus in the system.

B. Experiment 2 – Cumulative Fitness

Experiment 2 involves the maximisation of a cumulative objective function, where the score is computed by successively adding the inverse of the number of opinions. Here, the optimizer favors models which eliminate opinions as quickly as possible; essentially related to experiment 1, except that it is *maximising* a *cumulative* score.

Figure 6 presents the results for this sample run. As before, each generation is averaged over 20 separate executions. The optimizer is able to maintain a good diversity in population, as is demanded of evolutionary algorithms such as these. Even though very simplistic genetic operators were used, an increase in fitness is observed from maximum fitness values. While maximum fitness increases (albeit slowly), the mean fitness remains approximately the same.

In similar fashion to experiment 1, Figure 6 indicates a relatively quick improvement in fitness, which is met after approximately 10 time steps with what appears to be search stagnation. It is possible that the optimizer has simply reached a set of candidates with higest fitness possible and is oscillating between them (causing the variation towards the latter 80 generations). The first 20 frames is, however, convincing as to the optimizer's ability.

C. Experiment 3 – Turbulence

The third experiment attempts to induce some turbulence by favoring models which have a high standard deviation for a fitness value comprised of a distance to equilibrium among

 ISBN: 978-1-61208-390-2

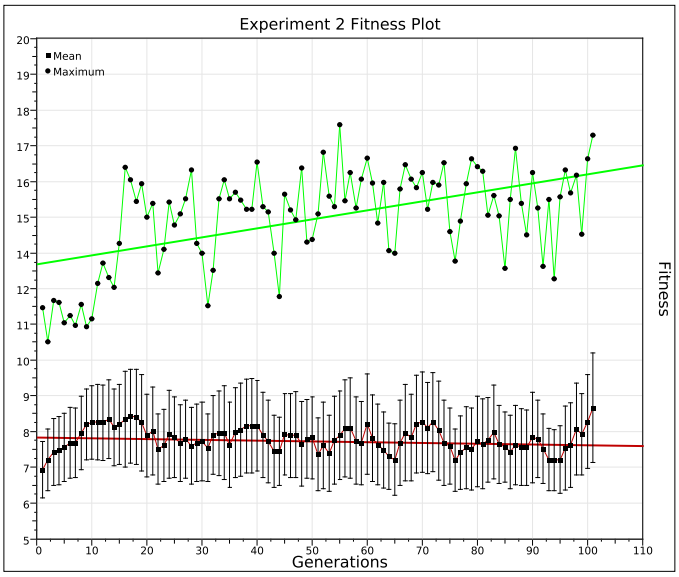

Figure 6. Fitness results by generation for experiment 2.

opinion counts. This equilibrium distance is computed by first histogramming the number of opinions in the lattice, and then summing the deviation of each opinion count from the number expected in order to form an equilibrium where every opinion has an equal share of lattice sites. This value is saved for every time step, and a standard deviation is computed on the very last time step, and added to the fitness score of the model in question. The objective function is therefore the standard deviation of the sequence:

$$s_i = o_c(i,j) - (o_n/16^2) \quad (1)$$

where $i = 1..n$, number of simulations is $n = 64$, $j = 1..o_n$, the number of opinions is $o_n = 32$, and $o_c(i,j)$ is the number of agents with opinion j in simulation (or candidate) i.

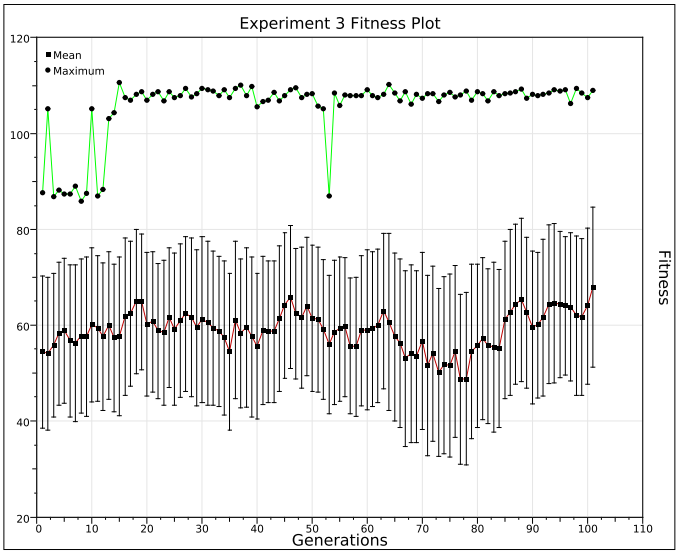

Figure 7. Fitness results by generation for experiment 3.

Figure 7 shows the result of this experiment as a fitness plot by generation. The purpose of the experiment is to determine the efficacy of the system as a whole, when dealing with significantly more complex objective functions. Shown in this figure is a sudden increase in fitness at approximately generation 15, which is reminiscent of the problem of local minima in metaheuristics [43].

To illustrate the effectiveness of the system, the best individual generated by the optimizer will be examined. The best program generated had a fitness score of 107, and had the optimisation construct composed of the following *k*-expression:

```
0 1 2 3 4 5 6 7 8 9 0
I0P0N1L0N1L0L0N0N0N2N2
```

For the purpose of the optimizer, the symbols are simply defined by assigning a type, and incrementing a number. Therefore, the first `if`-statement in Figure 4 is translated into `I0`, the second `if`-statement is translated into `I1` and so on, similarly for different statements. The terminal statements are identified by symbols which begin with `N` and `L`. `P0` is a manually inserted nonterminal, which simply executes its two arguments sequentially.

```
if (me − s) == 0 then
 select all
  idx = get_random_neighbour()
  s = getneighbour(lattice,idx)
 end
 propagate_single(newlattice,idx,me)
else
 propagate_single(newlattice,idx,me)
end
```

Figure 8. The best candidate generated to optimise the objective function provided for experiment 3. The *k*-expression for this code is I0P0N1L0N1L0L0N0N0N2N2.

When interpreted, the *k*-expression translates into the code shown in Figure 8. By inspection, this code does indeed provide turbulence. It is composed of a single if statement, which will replace the model code fragment in Figure 4 as part of the model code. Should the condition be true, then the randomly chosen neighbor shares the same opinion. If this is the case, then firstly, the simulation will choose a *different* neighbor, and propagate their opinion to that neighbor. Note that in the second case, the simulation does not check the opinion of the neighbor chosen. Should the condition be false, it signifies that the neighbor chosen does not hold the same opinion, and it would therefore propagate its opinion to that neighbor.

Since the lattice update style relies on a randomised-order update sequence between lattice timesteps, each cell is guaranteed to execute their program in a timestep. However, the final result does indeed depend on what order agents in the lattice are updated. Therefore, the result of the optimisation run may also depend on the update style. If it were a *monte-carlo* style update, then not all cells may be executed, and some may be executed twice.

The result shown in Figure 4 does indeed provide turbulence in the model by attempting to guarantee that an agent will propagate its opinion. It may be possible there is a program which provides more turbulence, however the optimizer in

 ISBN: 978-1-61208-390-2

this case was limited to three non-terminal statements in a candidate. In this example, only two were used: an `if`-statement, and a `P0` statement. More complex solutions may improve upon this fitness value, but do potentially require more computation.

V. Discussion

The three experiments conducted show that model programs can be optimised for different purposes. The model strategy that was being investigated here assumed that the modeller was attempting to use an optimizer to generate micro-behaviors that they are interested in. The results appear to show that it is indeed useful to obtain results which could be useful in a modelling situation. Exactly how useful the results are, would depend on the quality of the objective function, and terminals and non-terminals, which are discussed below.

There are unfortunately some caveats which are associated with Genetic Programming and general evolutionary algorithm literature [44], [45]. Particularly, the choice of terminals and nonterminals is a problem that is still applicable in the system discussed in this article. The implications of this mean that what is considered "optimal" by the optimizer, may in fact be severely limited by a wrong choice in initial program. The code shown in Figure 4 is very important, not in order and exact syntax, but more in terms of abstract states, lattice modifications, and state transitions. It is for this reason that future work will likely involve the design of a second DSL, which will handle finite state machines separately.

The optimizer also depends on the user for appropriate selection of parameters. Like many optimisation algorithms, a set of parameters is necessary. In the case of this system, probabilities of mutation, crossover and selection are predefined and hand calibrated. Clearly, some parameters would suit better in different situations. By considering extreme values in these parameters, it is easy to see how the algorithm would fail: setting the mutation probability to zero will remove all chances of injecting new material into the population of candidates, and therefore only "genetic drift" will occur [42], and similar problems will occur with the other parameters. Therefore, at least sensible generic values are absolutely necessary.

In particular, a parameter which is of particular importance here, pertains to the *k*-expressions of the candidate program population. One must choose a suitable head length for candidates, and then an expression length can be inferred from this to ensure that all generated programs are valid. This problem is loosely associated with the problem of avoiding code bloat in genetic algorithms [44].

At this point then, with the limitations of this approach, it may only be suitable to a small subset of models and their development. This is made clearer by the demand for an optimisation function, provided by the user. Such a function is not easy to formulate, and in many cases, generates results that appear to exploit a subtle flaw.

VI. Conclusion and Future Work

This article has introduced a proof of concept language and optimizer system presented in mid-2014, and provided some experimental results to indicate its efficacy. A model was written in this language, and a portion of "uncertain" code was also written with simplistic implementations of the Sznajd, Axelrod and Voter models. Three different optimisation functions were used, in order to instruct an optimizer on how to recombine the code given. The novelty in this approach lies in its use of a multi-stage language which is capable of run-time code generation using LLVM, and thereby avoiding costly run-time interpretations of code as many evolutionary algorithms do. Sample runs were made, and results presented in the form of fitness by generation plots.

The first experiment's optimisation function intended to minimise the number of opinions in the model. While each model was 16 cells by 16 cells, there were up to 32 opinions and 64 candidate models in the population. A limited improvement in minimum fitness was achieved. The second experiment attempted maximisation of a cumulative fitness function. It was intended that maximising this quantity would produce a model that attempts to reach consensus (minimum number of opinions) as quickly as possible. For brevity, a thorough examination of the best individual was omitted.

The third experiment involved a more complex fitness function, to determine how the system handles nontrivial objective functions. In short, an improvement in fitness was observed and the examination of the best individual provided some insights into the optimizer's behavior.

To conclude, the experiments appear to show a process that would be useful in a modelling situation, albeit, a tradeoff is presented in which the user must be able to provide a candidate solution, as well as an objective function. The quality of these directly influence the quality of the results, and if not adequately specified, may not be useful at all. These problems appear to stem from fundamental issues in the Genetic Programming and Evolutionary Algorithm literature.

Future work on this system involves a thorough statistical study by running the simulation on Graphical Processing Units (GPUs), and providing robust indications of both large-scale model optimisation, as well as small-scale mean performance. Further improvements on the optimizer itself is also under consideration.

References

[1] E. Bonabeau, "Agent-based modeling: Methods and techniques for simulating human systems," Proceedings of the National Academy of Sciences of the United States of America, vol. 99, no. Suppl 3, 2002, pp. 7280–7287.

[2] C. M. Macal and M. J. North, "Tutorial on agent-based modeling and simulation part 2: How to model with agents," in Proc. 2006 Winter Simulation Conference, Monterey, CA, USA, 3-6 December 2006, pp. 73–83, iSBN 1-4244-0501-7/06.

[3] C. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in SIGRAPH '87: Proc. 14th Annual Conf. on Computer Graphics and Interactive Techniques, Maureen C. Stone, Ed. ACM, 1987, pp. 25–34, iSBN 0-89791-227-6.

[4] G. P. Figueredo, P.-O. Siebers, and U. Aickelin, "Investigating mathematical models of immuno-interactions with early-stage cancer under an agent-based modelling perspective," BMC Bioinformatics, vol. 14, no. 6, 2013, pp. 1–38.

[5] G. P. Figueredo, P.-O. Siebers, U. Aickelin, and S. Foan, "A beginner's guide to systems simulation in immunology," in Artificial Immune Systems, ser. Lecture Notes in Computer Science, C. A. Coello Coello, J. Greensmith, N. Krasnogor, P. Liò, G. Nicosia, and M. Pavone, Eds. Springer Berlin Heidelberg, 2012, vol. 7597, pp. 57–71.

[6] R. Axelrod, "The dissemination of culture: a model with local convergence and global polarization," J. Conflict Resolution, vol. 41, 1997, pp. 203–226.

 ISBN: 978-1-61208-390-2

[7] F. L. Hellweger, E. S. Kravchuk, V. Novotny, and M. I. Glayshev, "Agent-based modeling of the complex life cycle of a cyanobacterium (anabaena) in a shallow reservoir," Limnology and Oceanography, vol. 53, 2008, pp. 1227–1241.

[8] J. M. Epstein and R. Axtell, Growing Artificial Societies : Social Science From the Bottom up., ser. Complex Adaptive Systems. Brookings Institution Press, 1996.

[9] J. M. Epstein, "Agent-based computational models and generative social science," Generative Social Science: Studies in Agent-Based Computational Modeling, 1999, pp. 4–46.

[10] J. Ferrer, C. Prats, and D. López, "Individual-based modelling: An essential tool for microbiology," J Biol Phys, vol. 34, 2008, pp. 19–37.

[11] V. Grimm and S. F. Railsback, Individual-based Modeling and Ecology. Princeton University Press, 2005.

[12] D. Helbing and S. Balietti, "How to do agent-based simulations in the future: From modeling social mechanisms to emergent phenomena and interactive systems design," Santa Fe Institute, Tech. Rep., 2011.

[13] K. Kacperski et al., "Opinion formation model with strong leader and external impact: a mean field approach," Physica A: Statistical Mechanics and its Applications, vol. 269, no. 2, 1999, pp. 511–526.

[14] K. Sznajd-Weron and J. Sznajd, "Opinion evolution in closed community," International Journal of Modern Physics C, vol. 11, no. 06, 2000, pp. 1157–1165.

[15] T. M. Liggett, Stochastic interacting systems: contact, voter and exclusion processes. Springer, 1999, vol. 324.

[16] D. Spinellis, "Notable design patterns for domain-specific languages," Journal of Systems and Software, vol. 56, 2008, pp. 91–99.

[17] W. Taha, "Domain-specific languages," in Pro. Int. Conf. Computer Engineering and Systems (ICCES), 25-27 November 2008, pp. xxiii – xxviii.

[18] E. Franchi, "A domain specific language approach for agent-based social network modeling," in IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, 2012, pp. 607–612.

[19] S. Tisue and U. Wilensky, "NetLogo: A simple environment for modeling complexity," in International Conference on Complex Systems, 2004, pp. 16–21.

[20] N. Collier, "Repast: An extensible framework for agent simulation," Social Science Research Computing, University of Chicago, Tech. Rep., 2003.

[21] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "MASON: A multiagent simulation environment," Simulation, vol. 81, 2005, pp. 517–527.

[22] W. Taha, "A gentle introduction to multi-stage programming," in Domain-Specific Program Generation. Springer, 2004, pp. 30–50.

[23] W. Taha and T. Sheard, "Multi-stage programming with explicit annotations," in ACM SIGPLAN Notices, vol. 32, no. 12. ACM, 1997, pp. 203–217.

[24] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: a multi-stage language for high-performance computing." in PLDI, 2013, pp. 105–116.

[25] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The evolution of Lua," in Proceedings of the third ACM SIGPLAN conference on History of programming languages. ACM, 2007, pp. 2–1–2–26.

[26] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua: An extensible extension language," Software: Practice and Experience, vol. 26, 1996, pp. 635–652.

[27] M. Pall, "The luajit project," 2008. [Online]. Available: www.luajit.org

[28] A. V. Husselmann, "Data-parallel structural optimisation in agent-based modelling," Ph.D. dissertation, Massey University, 2014.

[29] J. H. Holland, Adaptation in natural and artificial systems. Ann Arbor: University of Michigan Press, 1975.

[30] D. E. Goldberg, Genetic Algorithms in Search, Optimization & Machine Learning. Addison-Wesley Publishing Company Inc, 1989.

[31] S. van Berkel, "Automatic discovery of distributed algorithms for large-scale systems," Master's thesis, Delft University of Technology, 2012.

[32] S. van Berkel, D. Turi, A. Pruteanu, and S. Dulman, "Automatic discovery of algorithms for multi-agent systems," in Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion, July 2012, pp. 337–334.

[33] C. Ryan, J. Collins, and M. O'Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in Proceedings of the First European Workshop on Genetic Programming, ser. LNCS, vol. 1391. Paris: Springer-Verlag, April 1998, pp. 83–95.

[34] R. Junges and F. Klügl, "Evaluation of techniques for a learning-driven modeling methodology in multiagent simulation," in Multiagent System Technologies, ser. Lecture Notes in Computer Science, J. Dix and C. Witteveen, Eds. Springer Berlin Heidelberg, 2010, vol. 6251, pp. 185–196.

[35] R. Junges and F. Klügl, "Evolution for modeling - a genetic programming framework for SeSAm," in 13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, 2011.

[36] ——, "Programming agent behavior by learning in simulation models," Applied Artificial Intelligence, vol. 26, 2012, pp. 349–375.

[37] M. Privošnik, M. Marolt, A. Kavčič, and S. Divjak, "Construction of cooperative behavior in multi-agent systems," in Proceedings of the 2nd International Conference on Simulation, Modeling and optimization (ICOSMO 2002). Skiathos, Greece: World Scientific and Engineering Academy and Society, 2002, pp. 1451–1453.

[38] A. V. Husselmann and K. A. Hawick, "Towards high performance multi-stage programming for generative agent-based modelling," in INMS Postgraduate Conference, Massey University, October 2013.

[39] ——, "Multi-stage, high performance, self-optimising domain-specific language for spatial agent-based models," in The 13th IASTED International Conference on Artificial Intelligence and Applications. Innsbruck, Austria: IASTED, February 2014.

[40] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in Code Generation and Optimization, 2004. CGO 2004. International Symposium on. IEEE, 2004, pp. 75–86.

[41] C. Ferreira, "Gene expression programming: A new adaptive algorithm for solving problems," Complex Systems, vol. 13, no. 2, 2001, pp. 87–129.

[42] ——, Gene Expression Programming, 2nd ed., ser. Studies in Computational Intelligence, P. J. Kacprzyk, Ed. Berlin Heidelberg: Springer-Verlag, 2006, vol. 21, iSBN 3-540-32796-7.

[43] A. V. Husselmann and K. A. Hawick, "Levy flights for particle swarm optimisation algorithms on graphical processing units," Parallel and Cloud Computing, vol. 2, no. 2, April 2013, pp. 32–40.

[44] R. Poli, L. Vanneschi, W. B. Langdon, and N. F. McPhee, "Theoretical results in genetic programming: the next ten years?" Genetic Programming and Evolvable Machines, vol. 11, 2010, pp. 285–320.

[45] T. Weise, M. Zapf, R. Chiong, and A. J. Nebro, "Why is optimization difficult?" in Nature-Inspired Algorithms for Optimisation, SCI 193. Springer-Verlag, 2009, pp. 1–50.

 ISBN: 978-1-61208-390-2