ABSTRACT

Optimisation in the context of Agent-based Modelling has been thoroughly researched and reported in the literature. In particular, model parameter tuning has been done using a variety of parametric optimisers, and we are now entering a phase where agent behaviour itself is learned, not specified. The latter is proving to be problematic for a number of reasons. Algorithms earmarked for this purpose such as Genetic Programming and decision tree induction present their own problems. Defining the search space in terms of building blocks for these algorithms is surprisingly difficult. We propose a different methodology for accomplishing machine learning in the context of model induction. Instead of forcing the modeller to provide fine-grained and concise model building blocks, we provide a language where small portions of uncertain dynamics can be expressed concisely using domain specific knowledge. This has the potential to greatly increase the efficiency of building simulations for models, and reduce time spent on verification. Our language is built using recent concepts of multi-stage programming (MSP), providing run-time compiling and execution of code. This allows us to avoid the abstraction penalty. We provide detailed examples, and performance data for our implementation.

KEY WORDS
Agent-based modelling; domain-specific language; optimisation; model induction; multi-stage programming

1 Introduction

Many problems in Artificial Intelligence can be formulated in terms of an agent-based approach. Individual agents can be programmed in software to carry out relatively straightforward microscopically controlled actions that lead to complex emergent macroscopic behaviours that give great insights into large scale collective systems, that could not otherwise be modelled or described analytically. A simulations approach therefore is sometimes the only practicable manner to explore certain complex systems. However, the problem specification and encoding of such a system is both time consuming and error-prone. Furthermore, the resulting system even if correctly formulated by domain experts, may run so inefficiently or slowly on available computational resources as to restrict the size and time of simulated systems that can be explored. Agent-based modelling (ABM) systems can therefore benefit considerably from a domain-specific language approach to specification of the model and simulation code. In this paper we explore the use of a DSL prototype for ABM, in such a way that computational performance is not lost but may in fact be enhanced through use of appropriate practical optimisations.

Agent-based modelling refers to the use of locally interacting individuals (most often known as “agents”), the aggregate of which can be seen as a generative deduction of macro-level patterns, representative of a certain problem domain [36]. In many cases we refer to agents having spatial positions, but generally spatial positions of agents are not always necessary for exploring a model parameter space, as shown by models in supply chain management [43], for example, where it is the topology which is important.

Successful applications of the ABM technique include its use in disciplines and domains such as social science [8, 23, 38], medicine [15], vehicle routing [54], micro-

Figure 1. A visual representation of the optimiser evaluating 16 candidate models simultaneously. Each colour of agent is a different candidate.
Most authors [36] credit Schelling [50] for the first agent-based model in 1971. It is noteworthy that Schelling did not use a computer, but rather, paper and counters. Since the dawn of agent-oriented simulations there has been widespread disagreement on core terms and concepts [11], but this has subsided somewhat in recent works, since authors are generally more in agreement [36]. In the past, many authors defined their own terminology for once-off use [11]. From a practical standpoint, disagreement over ABM terminology stifles both understanding and progress. In general, agents are autonomous, situated and adaptive to their environment [36].

According to Macal and North, the process of ABM involves identification of the agents, theorising behaviours and interactions, followed by validation and analysis [36]. This often results in a trial and error process, as candidate behaviours are tested and parameters are calibrated. Following this process gives access to flexibility, elegant representation and effective representation of emergent phenomena [5].

Listing 1. An arbitrary example program with an uncertain construct.

```
sol
defvar c = select closest in group
do either
    move towards c by 0.1
    move towards {0,0,0} by 0.01
    done to minimise (distance to (c))
done
```

The success of ABM as a methodology largely resulted from the need to understand complexity. Preceding the era, complexity was undealt with, and terms such as “emergence” were even mystified [12]. As Epstein notes, the bond formation between an Oxygen atom and Hydrogen atom was taken by some to be fundamentally unexplainable [12]. Like the term “agent”, “emergence” is also contested, but is often used to refer to macro-level patterns which are not obviously synthesised by local interaction behaviour [13], even though this has also come under scrutiny more recently [12].

Despite the widespread success of ABM, there have been issues other than terminology also. There are many disciplines which do not teach programming as part of their curriculum. It follows that due to the simulation aspects of ABM, many scientists are not able to leverage the power of ABM [42]. Efforts to mitigate this issue have resulted in a plethora of software packages such as SWARM [37, 53], MASON [35], RePast [7], Netlogo [55], and others. For additional information on these, we refer the reader to the work of Railsback, Lytinen and Jackson [45]. Each of these systems has its own particular merits [45].

Due to difficulty in the modelling process, researchers have looked to other methods for streamlining the process. There has been a recent interest in using machine learning and optimisers in the search for local behaviours that lead to the desired macroscopic patterns [1–3, 29–31, 41, 52]. Issues associated with this are not trivial, but some are similar to that of parametric model calibration, particularly the choice of objective function [6].

Recent works of van Berkel [3] casts light on the use of Genetic Programming for generating models from building blocks. While van Berkel concedes that a good fitness function is a difficult challenge, not much emphasis is given on the choice of building blocks, other than being user-provided, and that some can be reused from other distributed algorithms. Both these problems are portrayed as open problems in Genetic Programming by O’Neill and colleagues [39]. In the works of Junges and Klügl, Genetic Programming is abstracted to a “learning core” which can conceptually be replaced with other techniques [30] making part of an overall methodology. While similar issues are dealt with in these works, the integration to the modelling platform SeSAm [32] simplifies the process greatly for the user. However, as SeSAm is based on the Java platform, programs (ie. models) generated at run-time pay a costly interpretation penalty.

Lessons learned from meta-optimisation [25] reaffirm that performance would be extremely problematic, mostly due to the averaging and re-averaging necessary to obtain a good fitness estimate. Van Berkel’s effort was distributed across a set of client processors, but performance results given indicated that the execution of a single program took upwards from 350ms for a program of lowest complexity [3]; together with averaging, the author reports total runtime of around three hours for one experiment. Recently, there has been strong interest in large-scale agent-based models, where a large number of agents are present [44]. Being able to represent large populations is sometimes a necessity. For example, in ecology, a technique was even developed for approximating the influence of multiple agents in a “super-agent” [49]. When referring to a population of agents, of the order of $10^{6}$+, it becomes impractical to use machine learning for developing models.

Our article is structured as follows: In Section 2 we present our approach in detail, and provide several short examples of syntax. In Section 3 we provide performance data, as well as an example usage. Finally we discuss the DSL and conclude in Sections 4 and 5 respectively.

## 2 Developmental Method

In this work, we provide a proof of concept study towards a high performance modelling support mechanism making use of a combinatorial optimiser to reduce provided uncertain constructs to concrete ones. We implement this through the development of a high performance domain-specific language [19–21] using a very recent multi-stage programming language named Terra [9]. To the best of our knowledge, we are not aware of other authors currently using multi-stage programming for the run-time generation
of candidate ABMs.

In our methodology, instead of the modeller providing fine-grained hypothesised local behaviours, the modeller instead provides an objective function and some possible behaviour within simulation code as part of the language. The difficulty in both these tasks are comparable: small variations in local behaviours as well as different choices of objective function can both lead to radically different results. However, the novelty in this approach lies in the elegance with which a model is prototyped. By allowing a modeller to express most of a simulation with reasonable certainty, and other parts with annotated uncertainty and clear objectives, an underlying optimiser can assist the modeller.

We combine a simple, high-level ABM programming language with a simple optimiser to construct a proof of concept for an enhanced modelling methodology. Recently, DeVito and colleagues published a multi-stage programming language named Terra [10]. The language makes use of LLVM and Clang [33] as well as Lua [27, 28]. The purpose of the language is to allow for high performance whilst still allowing the high-level elegance and simplicity of the Lua language.

In our implementation of this optimiser-assisted modelling language, we make extensive use of the high performance capabilities of Terra. The inherent optimisations of LLVM as opposed to the GNU C++ compiler [16] include automatic vectorisation among others, which is very desirable in our application. We use a very simple evolutionary algorithm to perform mutation, crossover and selection over a population of candidate programs. Each candidate program is evaluated separately, and effectively represents an entire ABM with homogeneous agents.

```sol
move awayfrom \{0,0,0\}
do either
  if distance to (c) < 5 then
    move towards c by mag*2
  else
    move awayfrom c by mag*2
  end

if distance to (c) > 5 then
  move towards c by mag
else
  move awayfrom c by mag
end
move towards (select furthest in all )
done to minimise (distance to \{0,0,0\})
done
```

Listing 2. An example program with a single-statement uncertainty.

Making use of the high performance behind Terra in a custom language is not entirely trivial. A special separate compiler is written in Lua, which constructs Terra statements using multi-stage operators. This additional compiler architecture includes a parser, a type checker, and a code generator. The Terra libraries provide good support for writing these, however. A sample program written in the custom DSL is shown in Listing 2, where an objective function is written on line 16. Essentially the parser will process this syntax, followed by type checking and then code generation. During the code generation phase, the compiler will emit Terra statements and expressions, which are combined together to form a JIT-compiled Terra function with the following signature:

```sol
terra(
    positions: &float,
    velocities: &float,
    scores: &float,
    frame: long
)
```

Should there be an uncertain construct in the model provided, the optimiser will iteratively improve on a population of models with regards to the objective supplied. Each model will have a user-supplied agent count, and therefore, there will be multiple models with multiple agents that must be evaluated. The Terra function shown above provides all the models to the compiled program. Each model is likely to run a different program, so the final Terra function will in fact be a collection of functions operating on separate parts of the arrays provided.

Scores (fitness values) are collected by the evaluation of the fitness function provided, which is added to a running total. This total is what is used by the optimiser for judging candidates.

The listing shown in Listing 2 provides for a single statement. In term of optimisation, this is not a difficult problem, provided that the objective function is well defined. Simply sweeping over the statements and averaging scores obtained will yield a good result. We added another statement to deal with multiple statements, where the order is also uncertain. This is shown in Listing 3. In this case, the output from the construct is multiple statements, in a certain order. Whereas the single-statement uncertainty uses a very simple optimiser, the multiple-statement construct uses a tailored evolutionary algorithm.

```sol
defvar c = \{-20,0,0\}
do some
  move awayfrom \{0,-200, 0\}
move awayfrom \{0,200,0\}
move awayfrom \{200,0,0\}
move towards \{0,0,0\}
move towards \{1,0,0\}
move towards \{-1,0,0\}
move awayfrom groupcentre
done to minimise (distance to (c))
vel = 0.9*vel
pos = pos + vel
```

Listing 3. An example program with a multiple-statement uncertainty.
Listing 3. An example program with a multiple statement, undefined order uncertainty.

To summarise the language, we present the overall process in Alg. 1.

Algorithm 1 The complete simulation process eliminating uncertain constructs.

- Terra custom parser reads agent description
- Read user-provided parameters
- Allocate & initialise space for $n$ candidates
  //small variations when no uncertainty is present
- Compute and compile a new generation of candidates
- Zero all scores
- while Termination criteria not met do
  for $x$ frames do
    Execute model programs
    Collect new scores into running totals by model
    Visualise the result
  end for
- Compute a new generation using collected scores
- end while
- Output best candidate in final population.

Termination criteria is simply a maximum number of generations at this point in time. During computation, every uncertain construct is replaced by a single, or multiple candidate statements depending on type of construct. Computing a new generation of programs is done by the process shown in Alg. 2.

Algorithm 2 The process of compiling a new generation of candidate models for evaluation.

- Collect scores
  for $x$ candidate models do
    Optimiser performs evolutionary operators
    Pass modified typed tree through code generator
    Wrap generated function code with arguments
    Emit wrapped code
  end for
- Overall generated code is compiled to machine code
- Fn pointer to compiled function passed to C++ via Lua

In our testing, we first compare straight frame computation of the Terra framework against an identical one that has been hand written in C++. We then provide a brief qualitative analysis of the practical usage of this language.

3 Selected Results

A visual representation of the evaluation phase is shown in Fig. 1. During performance testing, the entire simulation was recompiled with no visualisation, and strictly the computation time was measured. The C++ code used is shown in Listing 4 and the DSL code used is shown in Listing 5.

Although one might normally expect that code which has been automatically generated would be slower than hand written code, in this case we find our generated code is in fact almost an order of magnitude faster. The large difference in Fig. 2 is less surprising however, considering that the disassembly of the LLVM JIT compiled code yielded use of vectorised instructions, due to LLVM’s optimisation. All vectors in the DSL language are stored as Terra vector types, which in turn allows LLVM to do straightforward optimised code generation.

```c
float 3 goal = make_float3( 0.1f, 0.2f, 0.3f );
for ( int i=0; i < params->agent_count; ++i) {
  float 3 *mypos = (float3*)(h_positions + i*3);
  float 3 *myvel = (float3*)( h_velocities + i*3);
  *myvel = *myvel + (goal - *mypos) * 0.001;
  *mypos = *mypos + *myvel * 0.1;
}
```

Listing 4. C++ code used for comparison. The code uses some typedefs from CUDA.

```c
sol
  defvar goal = {0.1,0.2,0.3}
  move towards goal by 0.001
  pos = pos + 0.1*vel
done
```

Listing 5. Custom DSL code used to compare against GNU C++.

This is an interesting notion for future work - an automated system captures domain specific expertise but also optimisation expertise and experience for particular platforms. Although one could hand-optimise the C++ code with platform specific vectorisations and such like, this is labour intensive and error prone and is not likely to be feasible in the long term. It is greatly encouraging that one can actually attain better performance - as well as a more compact and easily attained complex systems implementation using the domain-specific language route.

As for an indication of how useful this DSL is, in a practical sense, we present a somewhat contrived example reminiscent of Reynolds’ Boids [47,48]. Consider the program shown in Listing 6.

The single agent-based model being described starts off with agents being given random velocities and locations. Then, through the query statement (reminiscent of the Proto int−hood statement), it is determined how many agents are in the neighbourhood of this agent. At this point, the neighbourhood size is hard coded. This program is executed by all agents, therefore, there is are intrinsic vel and pos globals. The closest agent is computed by a select state-
ment of the form select closest in group. We have arbitrarily defined the objective in this simulation to maximise the number of agents in the neighbourhood of each agent. Suppose then, that the only actions possible are those listed in the do some statement. At this point, the evolutionary optimiser is able to choose statements from these with replacement, but this will be modified in future. It is worth noting that while we are optimising in a similar fashion to Genetic Programming, we are only using terminals. This reduces the search space dramatically, but we envisage future applications where non-terminals may be necessary.

```sol
sol
defvar m = 0.001
defvar av_vel = {0.0,0.0,0.0}
defvar count = 1
query neighbours
count = count + 1
done
defvar c = select closest in group
defvar origin = {0,0,0}
do some
  if (distance to c) < 3 then
    move awayfrom c
  else
    move towards c
  end
move towards c
move towards origin
done to maximise(count)
```

```vel = 0.995*vel
pos = pos + vel
done```

Listing 6. A spatial ABM program written in the custom DSL, with selective uncertainty.

Following the uncertainty, the velocity is degraded and added to the position vector of the agent.

During testing, we did not average the fitness values, meaning that results from the evaluation stage were subject to considerable error. Evolutionary algorithms are very capable of dealing with error in objective functions, but we will be introducing additional accuracy once we complete integration of CUDA code generation.

Parameters used for various components of the language are shown in Tab. 1.

The optimiser yielded the code shown in Listing 7 for the do some statement. A screenshot of the model using this code block is shown in Fig. 4, whereas all the models together during optimisation is shown in Fig. 3. As the optimisation goal was to maximise the number of agents in the neighbourhood, it makes sense that agents should cluster together and not separate themselves. Of course, if it was necessary to ensure separation between agents, the first statement specified as possible behaviour could be placed before the do some statement.

```move towards origin
vel = vel + 0.5*av_vel```

Figure 2. Performance comparison between hand written GNU C++ code (above) and Terra (LLVM) compiled code (below.)
Table 1. Additional parameters used in the simulation and optimiser.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Steps per Generation</td>
<td>260</td>
</tr>
<tr>
<td>Number of Generations</td>
<td>10</td>
</tr>
<tr>
<td>Agents per candidate ABM</td>
<td>64</td>
</tr>
<tr>
<td>Number of candidate ABMs</td>
<td>32</td>
</tr>
<tr>
<td>Max magnitude of initial velocities</td>
<td>0.16</td>
</tr>
<tr>
<td>Max turn velocity</td>
<td>0.1</td>
</tr>
<tr>
<td>P(crossover)</td>
<td>0.8</td>
</tr>
<tr>
<td>P(mutation)</td>
<td>0.01</td>
</tr>
<tr>
<td>Box dimensions</td>
<td>30x30x30 units</td>
</tr>
</tbody>
</table>

4 Discussion

Our domain-specific language for spatial agent-based modelling shows promise as a proof of concept. The performance attainable for some test problems is particularly encouraging. The use of data structures such as Terra’s vector types - with a higher level semantics associated with them over plain low level arrays has allowed additional performance optimisation over that attainable from simple C++ coding and compilation.

In principle, other parallel and vector semantic information could be incorporated using for example OpenMP [40] or one of the other emerging directives based parallel annotation languages. There is considerable scope for work to extend the library to allow for other simulations to happen, and we intend to follow a suitable specification such as ASRM [46] for functionalities such as messaging.

Generally, designing languages is not an easy task, and often one pays the price for runtime interpretation of generated programs. Nevertheless, multi-stage programming is a very useful approach for generating programs at run time, and it is usually free of such overhead.

LLVM has been very successful and has a long history, so it appears that Terra can be expected to compile to clean and efficient machine code because of its use of LLVM. We have left much unexplored, such as the use of: inline C integration with Clang; further Lua integration; state variables; and more. We also would like to re-engineer the runtime environment to make use of a lattice instead, where vector computation is not so important. Fine grained data structures and associated algorithms lend themselves particularly well to the use of processing accelerators such as Graphical Processing Units [34]. We expect that the algorithms we have discussed in this present work can be appropriately ported to make good use of modern GPU devices and that this can be done within the context of back-end software libraries that support a DSL front-end interface.

5 Conclusion

We have prototyped a simple domain-specific language for agent-based modelling and have demonstrated its use for some optimisation problems. We have shown how this can address model development issues where a com-
plex agent-based model can be expressed more compactly and elegantly.

We have also shown the power and efficiency of the multi-stage programming approach and have found that through the use of appropriate tools and software technologies the achievement of good run time performance need not be mutually exclusive with a high level problem descriptive approach for this domain.

There is scope for adding considerably more power to the back-end library associated with our system, both in terms of its algorithmic repertoire but also in terms of its optimisation and platform capabilities. In particular we believe there is good scope for the addition of components optimised to run on accelerators such as graphical processing units (GPUs) to the run-time generated programs. We intend to use LLVM’s capability to generate CUDA code at runtime, which has subsequently been exposed by the Terra library. We will also extend the optimiser to parameter spaces with expressions such as between 0.1 and 0.2.

We have also focussed on using a set of terminal statements in uncertain statement blocks. We believe there to be scope to extend this to nested uncertain statements or otherwise, which would give rise to a non-terminal block and additional descriptive power.

Agent-based models will continue to find a diverse range of uses in optimisation problems and in other applications that need numerical rather than analytical solutions. The need to tackle ever more complex but also larger realistic sized problems that can properly explore multi-scaled phenomena will drive the development of systems like our prototype. High level elegant problem specification languages coupled with fast and efficient backend performance engines will open up use of these techniques to a broader range or real world uses of these artificial intelligence techniques.

References


[40] OpenMP: OpenMP Application Program Interface, 3.0 edn. (May 2008)


