ENGINEERING INTERNAL DOMAIN-SPECIFIC LANGUAGE SOFTWARE FOR LATTICE-BASED SIMULATIONS

Ken A. Hawick
Computer Science, Massey University
Albany, North Shore 102-904, Auckland, New Zealand
email: k.a.hawick@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

ABSTRACT
Modern high-level programming languages are making it practical to develop internal domain-specific languages (DSL) for even computationally intensive applications such as complex systems simulations. We present a prototype DSL to implement lattice-based operations for a whole family of simulation models on different lattices and with different neighbourhood localities. We use closure techniques to switch between lattice geometries and neighbourhoods at run-time. We show how a framework can be implemented in both C++ and Java and that a fluent interface DSL command-line layer and a graphical interface layer can both use this same framework. We find that we can drastically speed up the development and testing of a new simulation model that fits our DSL pattern and demonstrate this with significantly reduced lines of code needed for an incrementally added new model. We present results based on two-dimensional lattice models with discrete cell states for topologically square, hexagonal and triangular geometries and describe how these ideas generalise to higher dimensional problems. We also discuss how the use of internal domain specific languages and tools can facilitate further rapid development of other simulation codes and models.

KEY WORDS
domain-specific languages; software engineering; lattices; simulations; closures; fluent interfaces; Java

1 Introduction
One of the major goals for the designer of a computer programming language is to help the programmer express ideas and algorithms concisely and clearly. A major difficulty arises with even the most elegant programming language however, when programs become bigger and more complex. The standard computer science approach is "divide and conquer" or to abstractify ideas and component parts of a large program into software libraries that can be developed, tested and hidden away, to be invoked when needed. This lowers the amount of program code the programmer needs to hold on screen or in their mind at once and is the key to managing large-scale complex software development.

Modern programming languages have introduced many techniques and tools to help this abstraction and lowering of code complexity including: functions and subroutines; package modules; and of course classes and objects where operational code is combined with data structures to make abstract data types. So far so good, but even with all this apparatus, applications development programmers can still be easily overwhelmed by the sheer complexity and size of their programs [21].

One reason why there are so many varied programming languages [26] is that they all have different strengths, specialties and advantages for different sorts of applications. For example C is useful for programming operating systems and low-level device-drivers where access to the raw bits and bytes is important for efficiency. C++ allows a mix of low and high level ideas and is commonly used in systems and numerical applications. Prolog and its variants are still widely used for logic formulation problems. There are a number of declarative languages like Haskell and ML that aid in the development of artificial intelligence applications. Fortran is still a favoured numerical analysis language because of its efficient support for mathematical formulae. Java is attractive for graphical and web-oriented programming. Scripting languages like Ruby and Python are used for integrating together other programs. These mentioned are all general-purpose languages however and they carry with them a lot of operational baggage and idiomatic features that can obscure the essence of an application and hence sometimes make it unnecessarily complex to develop.

An exciting and relatively new technique is to produce a domain-specific programming language (DSL) [5, 6, 9] that provides a very high-level way for the programmer to formulate ideas for a particular application domain of knowledge and interest. The aim is to have the language focus concisely on just those concepts and aspects that are relevant to the particular problem domain and hide away any "boiler plate" code that is just an artifact of the underlying programming language. The domain in question could be a business problem using business jargon and language or for the discussion below it could use the scientific terminology particular to a problem in materials physics, phase transitions, and critical phenomena that are commonplace in the study of complex systems.

DOI: 10.2316/P.2012.790-043
A DSL can be implemented as either a full-blown language in its own right with all the necessary compiler [1] and builder environment apparatus to help the programmer [19]. Some of the computer algebra and problem-solving systems like Mathematica and Matlab use this sort of approach. A complete semantics is implemented to make high-level matrix-matrix and matrix-vector operations work appropriately. This approach takes a lot of development effort and to make a seamless programming environment a lot of other apparatus such as normal arithmetic, logic, text and string handling features must also be implemented. This approach is known as implementing an “external DSL” - it is external to the language that the system itself is implemented in.

However, often even these so-called external DSLs can be implemented to generate an intermediate program in a conventional language such as C that is then compiled in the normal way. The intermediate C code may never actually be seen or read by a human programmer, but this provides a powerful way to leverage off all the existing compiler and optimisation technological effort that went into for example the C compiler rather than redo all that effort for a new DSL.

Alternatively an even more light-weight approach is to use pre-processor technology or other features of an existing programming language to add on some macros or high-level language features so that the DSL features are effectively just pasted on top of the conventional language. This approach is known as an “internal DSL.“ This approach is familiar to C++ programmers whereby a package for handling complex numbers can effectively be added on to the language by overloading some of the existing arithmetic operators to add the necessary language semantics to handle real and imaginary parts of complex numbers. The relatively new Groovy language [3] provides a set of macro and extension features on top of Java so that much of the boilerplate code usually needed in a complex Java application can be automatically generated or at least minimised. Groovy in fact provides operator overloading extension features on top of Java so that much of the relatively new Groovy language [3] provides a set of macro and extension features on top of Java so that many simple DSL ideas are easily implemented.

The DSL approach [24] really comes into its own however when you are able to abstract some major set of operations and data structures together into a back-end library and have them invoked by the programmer through simple and compact programming language features.

The key issue for DSLs is that once a family of problems [4] is identified and one or two special cases are solved, then it's often feasible and efficient to develop a DSL framework [16] that effectively solves the whole family of problems rather than continuing to solve each one separately by hand. This makes it very productive to tackle a range of related problems to look for commonalities and patterns. Long-term problems such as the search for universal growth properties and characteristics of different critical and complex systems models, benefit considerably from the DSL technique. We can use DSL code-generation techniques to experiment with a different lattice geometry or with different model details relatively easily without the long, cumbersome and error-prone process of generating a new simulation code by hand for each special case.

This article is structured as follows: In Section 2 we summarise some key simulation models of interest and which comprise the specific domain for which we are attempting to engineer an appropriate DSL. In Section 3 we describe how to implement various lattices with a common storage scheme. We present some of the DSL and framework development techniques In Section 4 where we use Java to implement a neighbours framework and library using closures and a fluent interface. We present some selected results including model screen-shots a summary of the lowered model code complexity that arises from the DSL approach in Section 5. In Section 6 we provide a discussion of how these techniques might extend to other simulation models, lattice, and problem arenas. We offer some conclusions and areas for future work in Section 7.

## 2 Lattice Model Simulations Domain

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Application Arena</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ising</td>
<td>Magnetic Materials</td>
<td>[15]</td>
</tr>
<tr>
<td>Q-State Potts</td>
<td>Materials Science</td>
<td>[11]</td>
</tr>
<tr>
<td>Kawasaki</td>
<td>Materials Science</td>
<td>[17]</td>
</tr>
<tr>
<td>Sznajd</td>
<td>Opinions and Sociology</td>
<td>[27]</td>
</tr>
<tr>
<td>Axelrod</td>
<td>Culture Dissemination</td>
<td>[2]</td>
</tr>
<tr>
<td>Reichenbach</td>
<td>Cyclic Predator-Prey</td>
<td>[25]</td>
</tr>
<tr>
<td>RPSLS</td>
<td>Cycles and Parity</td>
<td>[13]</td>
</tr>
<tr>
<td>Rabbit-Fox CA</td>
<td>Predatory-Prey Cycles</td>
<td>[12]</td>
</tr>
<tr>
<td>Conway GoL</td>
<td>Complex System</td>
<td>[10]</td>
</tr>
<tr>
<td>Forest Fire</td>
<td>Ecology &amp; Damage</td>
<td>[7]</td>
</tr>
<tr>
<td>Eden</td>
<td>Cancer Growth</td>
<td>[8]</td>
</tr>
<tr>
<td>Epidemic</td>
<td>Disease Spread</td>
<td>[23]</td>
</tr>
<tr>
<td>Langton Ant</td>
<td>Complex Growth</td>
<td>[20]</td>
</tr>
<tr>
<td>Random Walk</td>
<td>Growth &amp; Information</td>
<td>[23]</td>
</tr>
<tr>
<td>Self-Avoiding</td>
<td>Constrained Growth</td>
<td>[23]</td>
</tr>
</tbody>
</table>

Table 1. Models that use discrete cell types of finite number of states and which can be modelled on the Bravais lattices in two dimensions and with different cell neighbourhoods.

The domain of applications ideas that this article focuses on is that of lattice-oriented simulations models. Table 1 lists some of the key models used in this work with a brief comment on their applications arena and a reference to a more detailed description. In brief these models all follow the pattern of requiring a set of scalar variables that comprise the model state to be initialised (usually in a random pattern) on a spatial lattice and then evolved according to some local update formula. Typically the evolution model involves a formula that specifies how an individual cell interacts with its local neighbouring cell and is hence updated based on its own state and that of its neighbourhood. Computational experiments using this sort of model
can reveal great insights into growth, scaling, information propagation, complexity and other emergent properties.

Algorithm 1 Common lattice model simulations architecture

set particular model to use
set model size \( N = L^d \), dimension \( d \) eg 2, geometry eg hexagonal
set \textit{model} neighbourhood size and geometry eg Moore
set \textit{geometry boundary conditions} eg open or periodic
set model parameters eg temperature
set random seed etc

\textbf{for all} \( N_R \) independent runs \textbf{do}
initialise/randomize model system/configuration
\textbf{for all} \( N_E \) equilibration steps \textbf{do}
apply simulation algorithm to whole model system
\textbf{end for}
\textbf{for all} \( N_M \) measurement steps \textbf{do}
apply simulation algorithm to whole model system
every \( m' \)th step period do a measurement
log measurements to file, or add to running statistics
save model configuration
\textbf{end for}
\textbf{end for}
gather measurement statistics

Algorithm 1 shows the typical outline of a computational simulation experiment where the core simulation algorithm would typically be expressed in a form like that of algorithm 2.

Algorithm 2 Common simulation model core algorithm for one time step

apply simulation model algorithm to whole system
\textbf{for all} \( N \) model cells) \textbf{do}
either do cells in order or randomly shuffled
gather relevant information \textit{from} neighbouring cells
apply the microscopic rule to each cell
propagate necessary information \textit{to} neighbouring cells
\textbf{end for}
if update was not “in place” then swap the buffer and current configuration

To make this architecture work we need to be able to parameterise the geometry and the neighbourhoods for an appropriate storage scheme.

3 Bravais Lattices & Cell Storage

The domain problem we focus on in this article is that of running simulation models on Bravais lattices in two dimensions. Table 2 lists the properties of the five Bravais lattices in two-dimensions as well as the triangular lattice, which we list separately for convenience. These five Bravais lattices [18] have different physical properties. However, from a software simulation perspective we need primarily only focus on the topology of the neighbours of individual sites as that is what will govern our choice of storage scheme and the necessary set of neighbour-gathering auxiliary routines needed. Topologically only the square, hexagonal and triangular lattice present different storage and indexing requirements.

While modern language support multidimensional arrays these are only really of any use for managing rectangular coordinate systems and furthermore the dimension of the problem needs to be fixed into the program source code. The dimension would normally dictate the number of array square brackets and indices used in a Java or C/C++ program for example. An alternative scheme that is much more powerful is to use a one dimensional storage scheme and to implement a mapping with appropriate auxiliary arithmetic routines to manage arbitrary dimensional and arbitrary lattice patterns.

We use a terminology with a \( d \)-dimensional system with length \( L_i \) in each of the \( i, i = 0, 1, 2...d − 1 \) dimensions, and with a number \( N \) individual sites. For a square lattice \( N = L^2 \) but the relationship is more complex for the other Bravais lattices. Most importantly for this work, the different models we experiment with use different preferred “neighbourhoods” which range from simple nearest-neighbour; next-nearest; Moore neighbourhoods and a radial proximity neighbourhood parameterised by some Euclidean distance of nearness. Not all of these apply to all our models but it is one of the goals of an integrated simulation system to be able to experiment with unusual or atypical neighbourhoods and geometries for the model family.

Figure 1 shows how a one dimensional storage mapping scheme can be used to implement the square (rectilinear); hexagonal and triangular lattices. The illustrations show the dual lattice patterns that arise from both the nodes and from the connecting edges. Note that the hexagonal and triangular are mutual dual lattices of one another, whereas the square is its own dual lattice.

A Bravais lattice [18] in 2 dimensions is defined by the in-principle infinite set of discrete points generated by the translations:

\[ \mathbf{R} = n_1 \mathbf{a}_1 + n_2 \mathbf{a}_2 \] (1)
where \( n_1, n_2, \in W \) are integers and \( a_1, a_2 \) are the primitive vectors that define the principle axes of the lattice. By definition a Bravais lattice appears identical from any position \( R \). There are five Bravais lattices in 2d: square, rectangular, oblique, rhombic, and hexagonal. Topologically the square, rectangular and oblique lattices can be treated the same, and the relationship between the four nearest neighbours of each point is the same. The rhombic lattice still has 4 nearest neighbours but the topological relationship is changed by the “face centering” notion. The hexagonal lattice likewise must be treated differently and it has 6 nearest neighbouring points for each site. It is often convenient to denote a triangular lattice which has 3 nearest neighbouring points. The triangular system must be stored differently to ensure compact topological space usage and strictly speaking the triangular lattice is no longer a Bravais lattice because of the even/odd asymmetry.

In terms of developing a software library and associated data structure, it is convenient to adopt a one-dimensional storage mapping index. We refer to this as a k-index and our software will map a unique integer \( k \) to each cell in an actual finite lattice.

This places some relatively minor constraints on some of the lattice structures, and a useful general constraint is that the lattice lengths are even so that:

\[
L_x = 2m_x, L_y = 2m_y, m_x, m_y \in W
\]

These condition means that all the topological arrangements shown in Table 2 can be used with wrap-around periodic boundary conditions as well as open ones.

Any actual simulation we perform must of course use a finite lattice model and associated array of points. In practice for many of the model systems of interest the interactions are finite in range and therefore as long as each point has the same number of neighbours then locally we achieve symmetry. This can be done by using periodic or wrap-around boundaries. These will affect long range properties of the model but not short range ones.

4 Implementing a Lattice-Oriented DSL

The main goal is to find a way to implement software that closely matches the algorithmic ideas discussed above and in particular the architectural pattern in Algorithm 1. In practice we want a high level interface that the user/programmer can work with to formulate simulation experiments with individual models, while being able to set model parameters easily and concisely. In support of this we need a suitable library framework that supports all the necessary simulation operations, but that is extensible and flexible enough to support non-trivial adjustments such as supporting arbitrary lattice geometries and arbitrary neighbourhood ranges that will be set at run time.

Figure 2 shows a traditional idiomatic form to read
Figure 2. Animat Model Command-Line Argument parser for use in eg a Unix shell script.

flag or switch-like command line settings from a Unix/DOS shell program. This extremely simple “language” of settings can be used to adjust the parameters for a fixed pattern of operations as described in Algorithm 1. It is limited however in its expressiveness and it is not feasible to mix simulation, equilibration and measurement loops and so forth. A much more powerful form is that of the fluent interface, which can be implemented for this sort of problem using method-chaining in a language like Java.

Figure 3. Internal DSL Fluent Interface for AnimatModel Lattice Simulation in Java using method-chaining.

The code listed in Figure 3 shows the sort of fluent interface internal DSL programs that were used to generate the screen-shots in Figure 6 in Section 5. The model object is created using a factory method (New()) and a sequence of chained setter methods can initialise the parameters. These can be done from string arguments but in this context it is more elegant, and clearer to implement symbolic pseudo-types such as HEXAGONAL, PERIODIC, ISING and so forth. These can trigger the injection of an appropriate pseudo closure to arrange the correct neighbourhood list method be called to gather neighbours of the lattice sites. This can be easily done using for example the enum enumerated type mechanism in Java. Various command-oriented chain-able methods can be implemented within the framework. For example the step() method is used to time-step the chosen model using the particular evolve() method that has been injected using the pseudo-closure mechanism discussed below. Similarly analysis methods to count the component cluster in the model or other statistical analyses methods can be invoked from the internal DSL methods.

The particular power of an internal DSL approach is that apparatus for managing loops for example and any other arithmetic or logic operations are available from the host language - in this case Java. This saves considerable time and effort in reimplementing and such operations in the embedded internal DSL. A number of useful mechanisms in modern Java can be exploited to make it easier and elegant to implement the embedded DSL.

Figure 4. Inserting a closure with evolve(int) method into HashMap of models – example shown is for the Ising \( Q = 2 \) state model.
Could share with Eden
Could share with Ising
Also implements Prisoner
Could share with Random
Comes free from Ising

Table 3. Lines of Java Code unique to each model, supported by \( \approx 3,700 \) lines of library code in the Animat-Model suite.

### 6 Discussion

Our system required an identification of the software algorithmic and data structural patterns that are common to this family of lattice-based simulations. These ideas have been arrived at over several years of implementing and re-implementing individual simulation model programs. The distillation of these ideas in a unifying implementation using the DSL techniques presented here provides a satisfying way to develop new models and particular hybrid models that might be otherwise too complex to implement.

The Java language has proved a good compromise for developmental purposes. Ideas like symbolic types for models and neighbourhood specification can be implemented with enum enumerated types. Many of the key operations are iterations over cells or over neighbourhoods and Java’s for-each mechanism aids the clarity of these. Java does not have a properly implemented closure mechanism but anonymous inner classes that close over variables in scope and inject appropriate concrete methods into a data structure at run time provide a pseudo-closure mechanisms that has been satisfactory for the system reported here.

Higher-level semi-dynamic languages like Groovy that sit on top of Java do provide a closure mechanism and indeed nested-closures and therefore offer promise for a higher level implementation of these ideas. However, there is still attraction in a system that will not sacrifice run time performance and being able to work with framework code that is easily implemented in both Java and in C/C++ is useful from this perspective.

We have not presented details on a model-view-controller graphical interface to the system. This was readily implemented in Java’s Swing graphical library. Graphical widgets such as slider, buttons and checkboxes can be easily set up to use the underpinning framework described in this paper. The method-chaining methods can quite eas-

```
public abstract class Neighborhood{
    public abstract int[] list(int k);
    public abstract int at(int k, int n);
    public int neighbourHoodSize(int k){ return list(k).length; }
    public int randomNeighbour(int k){
        return at(k, rng.nextInt(neighbourHoodSize(k)));
    }
    public int randomSite(){ return rng.nextInt(N); }
}

Neighbourhood neighbourHood;
if( nearestNeighboursOnly ){ // inject appropriate closure
    neighbourHood = new Neighborhood(){
        public int[] list(int k) { return neighboursF1(k); }
        public int at(int k, int n) { return aNeighbourF1(k,n); }
    }
    . . .
}
else if( nextNearestNeighboursOnly ){
    neighbourHood = new Neighborhood(){
        public int[] list(int k) { return neighbours2F1(k); }
        public int at(int k, int n) { return aNeighbour2F1(k,n); }
    }
}
else if( moreNeighbourhood ){
    neighbourHood = new Neighborhood(){
        public int[] list(int k) { return neighboursMF(k); }
        public int at(int k, int n) { return aNeighbourMF(k,n); }
    }
}
else if( radialProximitiesNeighbourhoods ){
    neighbourHood = new Neighborhood(){
        public int[] list(int k) { return proximities[k]; }
        public int at(int k, int n) { return proximities[k][n]; }
    }
}
```

Figure 5. Inserting Closure with list(int), at(int,int) methods to determine neighbourhoods.

### 5 Selected Results

Table 3 shows a conservative (ie an upper bound) count of the lines of unique Java code (LOC) for each of the models implemented. These are conservative estimates and include comments, spacing and so forth. They compare very favourably with estimates LOC values given in [14] which were typically four to five times greater in each case than the values given here.

Some of these models are illustrated in the screenshots of Figure 1 which shows system run on square, hexagonal and triangular lattices.

The main outcome of the work reported here is a framework and internal DSL implementation that allows new simulation model to be invented and quickly implemented for experimentation. The key result from a software engineering perspective is therefore to be able to implement a new model in as few lines of elegant and human-readable code as possible.
Figure 6. A selection of screen-dumps from models using different lattices – Top row: Square 128x128; Game-Of-Life; Random Walk; Langton Ant, Middle row: Hexagonal 128x128; Epidemic; Ising; Reichenbach, Bottom row Triangular 128x80; Ising, Potts (Q=4); Forest Fire.

We have very much focused on implementing an internal DSL in this work. The method-chaining approach gives a human-readable fluent interface that is close conceptually to the domain-specific jargon of lattice simulations. It is also straightforward to implement using Java and makes available looping and arithmetic facilities of Java to the DSL simulations user without requiring reimplementation. Nevertheless it may be worthwhile to experiment with an external DSL that can drive a parser generator to generate code in other high-level language aside from Java. This is a potentially powerful route to incorporate automatic parallelism into the implementations [22].

data parallel or multi-threading parallelism ideas can be implemented according to a strict pattern that will work well for this class of regular lattice simulation model. A code generator has the potential capability to generate almost all the boilerplate parallel code to support this. Hiding this from the ordinary simulations domain user is attractive.

7 Conclusion

We have described how an internal domain-specific language can be built to simplify running computational experiments with lattice-oriented simulation models. We have implemented the DSL using Java and some of its modern features including type-parameterised generic data structures; for-each iterators; enum enumerated types for internal symbols; and a closure-like approach to parameterise lattice geometry and choice of model.

We have been able to develop a relatively short (≈ 3,000 – 4,000) line code library framework that enables new models to be developed with incremental new code of only a few tens of lines of Java. This compactness makes
engineering software for models and especially for hybrid and new models much faster and easier to check, test and therefore increase confidence in the results quickly.

Java was used primarily although much of the underpinning framework code was written in a manner that was implementable in C/C++ or any similar modern programming language. The internal DSL technique adopted used method-chaining to provide a fluent interface but there is also scope to consider an external DSL implemented with an independent parser/generator that would facilitate independence from Java.

Another approach would be to use an even higher level language such as Groovy which effectively sits on top of Java to provide proper closures and other associated facilities that we have had to emulate with Java. This has the potential to lower code length and hence code complexity even further and facilitate a semi-interactive addition of new models to the system.

The models described have been restricted to integer cell types on Bravais lattices in two dimensions. There is scope for extending the system to cope with higher dimensions, and to models that use floating point, complex or hybrid types such as matrices or tensors.

Domain-specific language development for what would otherwise be quite lengthy numerical applications codes like the ones here offers a significant step forward to engineering hybrid and complex simulations software, with considerable savings in programmer time and effort possible.

References