Computational Science Technical Note **CSTN-123**

Engineering Domain-Specific Languages for Computational Simulations of Complex Systems

K. A. Hawick

2011

Simulating complex systems often requires high-performance software to be custom developed. However families of simulation problem do arise in which many computational tools, data structures and semantic ideas are shared in common. Software engineering mechanisms that support the sharing and reuse of developed code are strongly desireable to save cost on development effort, testing and validation effort, and specifically to lower the time involved from concept to implementation for a new simulation model. Domain-Specific Languages (DSLs) provide a valuable such mechanism and act as a semantic bridge between application domain expert and programmer. Even in cases of computational scientists where these two roles might be played by the same individual, a DSL provides a powerful mechanism to abstract domain-specific ideas to improve the development time by lowering the software code complexity. We discuss some families of complex systems simulation problem and some experiences with domain-specific languages and tools – particularly when applied to manage simulations where parallell and distributed computing technologies are used to speed-up the process of statistics collection from the numerical experiments.

Keywords: domain specific languages; simulation models; complex system; software engineering

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-123,
        author = {K. A. Hawick},
        title = {Engineering Domain-Specific Languages for Computational Simulations
                of Complex Systems},
        booktitle = {Proc. Int. Conf. on Software Engineering and Applications (SEA2011)},
        year = {2011},
        number = {CSTN-123},
        pages = {222-229},
        address = {Dallas, USA},
        month = {14-16 December},
        publisher = {IASTED},
        doi = {10.2316/P.2011.758-046},
        institution = {Massey University},
        keywords = {domain specific languages; simulation models; complex system; software
                engineering},
        owner = {kahawick},
        timestamp = {2012.01.15}
}
```

Technical Report CSTN-123

Engineering Domain-Specific Languages and Automatic Code Generation for Computational Simulations of Complex Systems

K.A. Hawick
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
Email: k.a.hawick@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

September 2011

ABSTRACT
Simulating complex systems often requires high-performance software to be custom developed. However families of simulation problem do arise in which many computational tools, data structures and semantic ideas are shared in common. Software engineering mechanisms that support the sharing and reuse of developed code are strongly desirable to save cost on development effort, testing and validation effort, and specifically to lower the time involved from concept to implementation for a new simulation model. Domain-Specific Languages (DSLs) provide a valuable such mechanism and act as a semantic bridge between application domain expert and programmer. Even in cases of computational scientists where these two roles might be played by the same individual, a DSL provides a powerful mechanism to abstract domain-specific ideas to improve the development time by lowering the software code complexity. We discuss some families of complex systems simulation problem and some experiences with domain-specific languages and automatic code generation (ACG) tools – particularly when applied to manage simulations where parallel and distributed computing technologies are used to speed-up the process of statistics collection from the numerical experiments.

KEY WORDS
domain specific languages; automatic code generation; simulation models; complex system; software engineering

1 Introduction

Developing computer simulation programs for studying complex systems models is of continuing importance in computational science and engineering. High performance simulations that support computational experiments and the study of phase transitional behaviour, emergence, complexity and logarithmic scaling phenomena in models are critical tools for understanding many systems that are not directly accessible to either analytical theoretical methods nor to physical experimentation.

Engineering the software [1, 2] for such simulations has traditionally been done by hand and although many excellent problem solving and modelling environments and tools have become available they are typically unable to tackle the very large system sizes and long simulation times that are necessary to carry out statistically meaningful experiments. Often in the past a new simulation code for a single model might take up a "whole PhD's worth of effort" in terms of code development and testing prior to being able to carry out production quality computational experiments. This is exacerbated in the case of complex simulations algorithms where techniques and ideas from several different disciplines must be brought to bear with an ensuing increase in code complexity to bolt together separately developed and sometimes structurally incompatible code components [3]. The advent of high level programming languages and associated tools has helped, but an alternative approach is becoming feasible – the use of domain-specific language (DSL) and automatic (high-level) code generation (ACG) techniques.

An area of importance in computational science is the categorising and classification of different models into their universality classes – identifying what properties they share and what are their essentially unique qualities. This work requires development of not one but many simulation models. Fortunately the central notion of DSL development is very well suited to this problem. DSLs are particularly useful to drive development of families of programs or "product lines" that have some pattern or shared commonalities [4–6]. DSLs tend to be easier to develop once the patterns and shared properties have been fully identified. It is less easy to apply them to an area of work in progress , where the model properties are still being discovered. Nevertheless ACG techniques and tools that tend to be closed linked to DSL tools can provide a platform to greatly accelerate the development of simulation model programs.

An important article by Spinellis [7] described some of the main known patterns of usage for DSLs. Although ideas such as language-oriented programming [8] have been known and reported in the literature since 1994, DSL development and deployment is still a relatively new area with most reported activity in the literature only over the last 10-12 years [9–12].

DSLs and associated techniques have been employed in a range of different application areas [26]. Their use is still not completely widespread although this is likely to accelerate as better tools become available and more experiences are reported. In addition to their use in generating programming languages and tools themselves [27, 28], areas of reported successful DSL/ACG use to date include: communications and telephony [29, 30]; real-time- embedded systems [31]; digital circuit design [32] and field programmable gate array device deployment applications [33]; distributed and computational grid applications [34]; and mathematical [35] and equation-based problem formulation [36–38] . Parallel computing [39] is also an area of further promise for DSL and ACG.

DSL tool development has continued [40] and there is renewed interest [41] in the approach largely due to improvements in language and code generation tools [42–44] and DSLs may now have practical as well as idealised advantages over the use of general purpose programming languages [45] A powerful recent tool for DSL development [46] is Parr's ANTLR tool [47], and the StringTemplate tool [48] that underpins code generation in ANTLR is one of the tools we use in the work reported in this present article.

This article describes some prototypes and early successes in applying DSL and ACG techniques to a very specific sort of simulation program for studying complex systems models. We describe an emerging architecture for a developer-driven ACG tool and how it has been used to automatically generate skeletal simulation codes in the language C++. We make use of a number of programming languages and associated tools including the StringTemplate system upon which much of our code generator is based. We also use Java and its Swing Graphical User Interface libraries to develop the user-interface and forms that drive our generator. We are able to integrate generated skeletal programs together our own domain-specific application libraries and to reduce the size and complexity of simulation programs from that of hand-generated and optimised codes, and for a number of different models.

We use our prototype platform and the experiences gained using it to discuss what is essentially and instance of to employ model-driven application development in computational science. Our article is structured as follows: In Section 2 we summarise some common properties and patterns of the simulation models we study. We give a description of our code generator framework ideas in Section 3 with a focus on implementation details in Section 4. We give some results concerning the reduction in the number of lines of code for generated versus hand-crafted simulations in Section 5 and discuss the code complexity reductions and associated practicalities of the approach in Section 6. We conclude with a summary and some areas for further development in Section 7.

Model Name & Citation	Geometry	Cell Type	Neighbourhood	Algorithm	Parameters
Ising [13]	1,2,3 D	Bit	NN	Stoch.	T
Q-State Potts [14]	1,2,3 D	$\lceil \log_2 Q \rceil$ bits	NN / N-NN	Stoch.	Q, T
Classical XY (Clock) [15]	1,2,3 D	float pair	NN	Stoch./Time-Integn.	T, δt
Classical Heisenberg [16]	1,2,3 D	float triple	NN	Stoch./Time-Integn.	T, δt
Eden growth [17]	2,3 D	Bit	NN	Stoch.	p
Spatial Epidemic [18]	2,3 D	bit pair	NN	Stoch.	p
Wolfram Automata [19]	1 D	Bit	NN	Determ.	rule No.
Game of Life [20]	2 D	Bit	Moore	Determ.	p, B-S rules
N_S species RPSLS [21]	2, 3 D	$\lceil \log_2 N_S \rceil$ bits	NN / N-NN	Determ.	p-vector, N_S
Predator-Prey [22]	2, 3 D	Bit	NN	Stoch.	various
Lotka-Volterra [23]	1,2,3 D	float pair	NN + N-NN	Time-Integn.	Popn. ratio, δt
Cahn-Hilliard [24]	2, 3 D	float	up to 4'th-NN	Time-Integn.	T, M, δt
TDGL [25]	2, 3 D	complex	up to 3'rd-NN	Time-Integn.	b, c, δt

Table 1: Some rectilinear geometry simulation models that share a lot of "boilerplate" and framework code.

2 Simulation Model Commonalities

Many of the physical, engineering and other scientific areas of interest that exhibit phase transitions and temporal growth or decay have a great deal in common with one another terms of their practical implementation. Examples include the Ising model of a ferromagnet; the Potts, Clock and Heisenberg models of spin systems; classical cellular automata (CA) such as the Wolfram CA or Conway's Game of Life and variations of it such as the Zombie CA model and cyclic Rock-Paper-Scissors-Lizard-Spock (RPSLS) models. Other growth systems such as the Eden and epidemic models and various diffusion models such as the self-avoiding walk can also be studied on similar geometries with similar coding structures. More sophisticated spatial agent-based models such as Predator-Prey systems have similar properties and are also based on a small number of variables stored at each site of a mesh or lattice. Individual or systems of partial differential equations such as the Lotka Volterra species model; the Cahn-Hilliard field equation governing materials segregation and the time-dependent Ginzberg-Landau field equations governing super-conductivity behaviour can also be expressed in a similar simulation code framework.

These models all require the essential approach of: initialise a mesh of cells; time evolve or randomly evolve each cell based in the values of its neighbouring cells; measure some macroscopic properties; repeating until equilibrium is reached or until the transient stage is over.

Table 1 shows some of these typical simulation models and their shared properties, indicating that an automated code generation approach is useful. The models shown share a lot of common "boilerplate" code to handle parameters, data management including configurations, user interfaces as well as many common more application domain-specific aspects such as code for: model initialisation; time evolution (deterministically, stochastically or by time-integration); measurement; statistical averaging and moment calculations; and reporting.

3 Code Generator Framework

Looking more closely into the shared patterns and code fragments and data structures we can summarise the simulation model meta-algorithm as follows:

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

A great deal of these details are expressible in program source code that is essentially "boilerplate" – that is code that is almost identical for different simulations.

The geometric arrangement of degree of freedom variables is often handled very similarly regardless of

Algorithm 1 Common numerical simulations architecture

```
set model size $N$, dimension $d$, geometry
set simulation duration
set microscopic model parameters
set random seed etc
for all $N_R$ runs do
    choose a random seed/start point etc
    initialise model system/configuration
    for all $N_e$ equilibration steps do
        apply simulation algorithm to whole system
    end for
    for all $N_M$ measurement steps do
        apply simulation algorithm to whole system
        every $m$'th step period do a measurement
        log measurements to file, or add to running statistics
        save model configuration
    end for
end for
gather together the measurement statistics
```

Algorithm 2 Common numerical simulations core algorithm

```
apply the simulation algorithm to whole system
for all $N$ degrees of freedom (eg model cells) do
    either do cells in order or randomly shuffled
    gather relevant information eg neighbouring cells
    apply the microscopic rule to each cell
end for
if update was not "in place" then swap the buffer and the current configuration
```

the simulation model details. The rectilinear hyper-brick management code for example, can be used in many cases of completely different models as it manages the gathering of neighbouring cells, the boundary conditions and the saving and loading of a rectilinear data structure to/from file or other persistent storage medium. The microscopic rules that differentiate one simulation model from another may in fact only require a few lines of code in a language like C or C++. It is therefore worthwhile considering how the rest of the coding apparatus can be reused or even automatically generated.

It is particularly interesting if we can change the simulation model geometry - eg from a line to a square mesh to a cubic mesh or even to hexagonal or triangular meshes in two dimension to face-centred or body-centred cubic geometries in three dimensions, or even arbitrarily dimensional hyper-cubic geometries. It is very attractive to be able to engineer a single unified simulation geometry code apparatus that can handle all these cases without separate program source codes needing to be generated and maintained for separate special cases. This can be factored out into a library that makes use of a single dimensional array of memory but where the spatial dimensions are managed separated and coded into a single k-index. This means that the geometric dependence can be specified entirely at program run time and that the simulation program cane be written as a single source code structure that handle all dimensional cases.

The neighbourhood of a cell is often a key aspect of a particular simulation model. Many models use simple nearest neighbours but some use the Moore neighbourhood which incorporates first and second nearest neighbours. Some models based on partial differential equation finite-differencing stencil operators need more distant neighbourhoods such as third or fourth neighbours. Specification of a neighbourhood is usually a compiled-in property - it is fundamental to the common understanding of a model so is not something that needs to be controlled by the user of the simulation program at run time. It does need to be specified by the developer however – usually by a choice of library call.

Not all features can be easily factored out into libraries. Different models have different algorithmic requirements. Some can be expressed as a deterministic sweep over all lattice sites in order. Other models may require some randomization of the order of sites to update to avoid introducing artificial artifacts into the solution. Some models require a multi-phase update to ensure some conservation properties are properly satisfied.

Models also vary in the fundamental type of the individual lattice cells. This might vary from a single bit to an integer, double precision floating point number or a complex number or even a vector of field values. The fundamental data type for a (speed optimised) simulation program is usually something that has to be determined at compile time.

Model will have different parameters that control them. Typically we want the user running experiments to be able to defer choices of all these to run time. It is common to employ a Unix command line or shell script run time paradigm such as embodied in:

```
#!/bin/sh
#  shell script to run mysimprog
for TEMP in 0.1 0.2 0.3 0.4 0.5 0.6
do
  mysimprog -T ${TEMP} -p 0.5 -L 1024
done
```

Figure 1: Typical shell script approach to running a simulation program where a loop iterates over parameter values, passed in through the OS/shell argv-command interface.

Figure 1 shows a shell script iterating over some parameter - in this case a double precision temperature, and passing that into a Unix style C/C++ compiled program using a command line optional argument. This is a common and powerful way to structure speed optimised simulation programs for numerical computational experimental studies.

4 SPG Implementation

To support this paradigm of simulation program we want to be able to semi automatically generate the C/C++ source code for individual simulation models in a way that is model-driven. The “boilerplate” code that is needed to manage parameters, initialise variables, manage files, collect statistics, is all common, is well-understood and is tedious but error-prone to write separately for every new model case.

The approach taken is to combine a graphical interface tool that allows a user to edit various properties and specify which libraries s to incorporate, which parameters will be specified at run time, which parameters can safely be compiled-in, which library data structures will be used, which files will be written and so forth. The main point is that for a family of programs such as arose from the category of simulation models being targeted, there is much in common and many – but not all – ideas can be factored out into reusable libraries. The approach is to factor out what can be, but still allow the developer to fill into speed optimised C/C++ code for the model-specific details.

Figure 2 shows a schematic of the architecture of the SPG simulation program generator that subsumes much of the commonalities of this family of simulation programs. The user interactively prepares properties and attributes of the planned simulation program prior to automatically generating skeletal C+++ program source code. That code can then be hand customized and augmented for those highly application-specific details that were not worth while encoding up as templates.

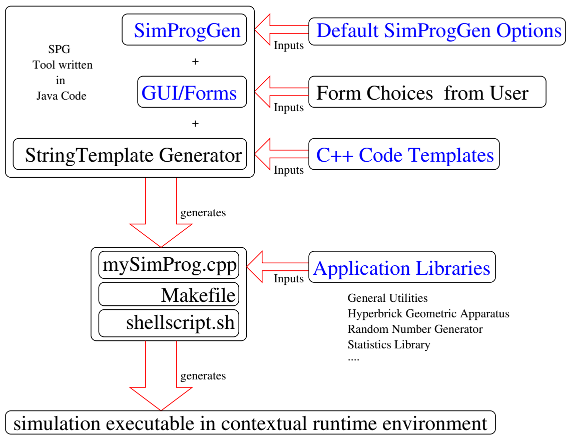
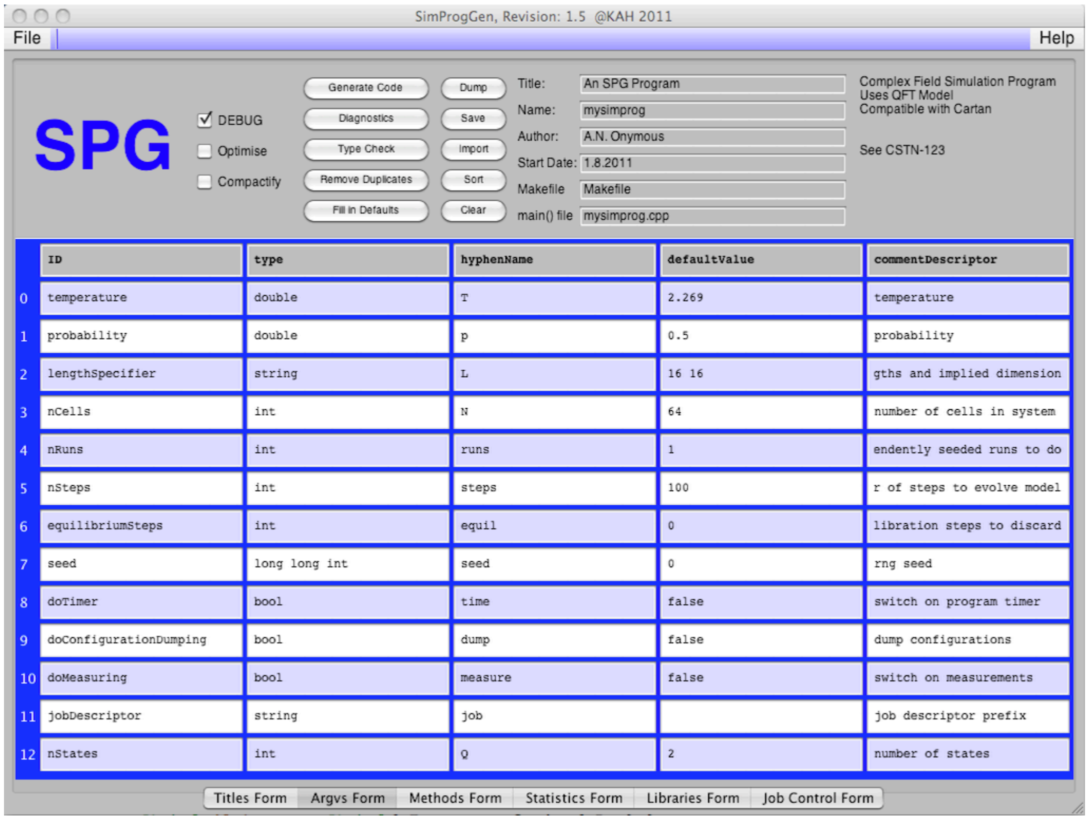


Figure 2: SPG SimProgGen Architecture showing downwards generation flow augmented by user supplied information from right to left.

Figure 3 shows a screen-dump of the prototype SPG tool. The table view shown is setting up the command line arguments of a simulation program ready to be generated. The user can specify a default set to be mixed in from a YAML file. These can be edited and check within the graphical form, before packaging for injection into the StringTemplate code generating templates. YAML markup is very much briefer, simpler and more human readable than other formats such as XML. YAML is adequate for the relatively small parameter and customisation property files we require. YAML uses ‘hyphens to denote sequences or lists, and colons to denote maps. These conveniently map to ArrayLists and HashMap classes in Java and we can therefore use YAML apparatus to load and save our serialized property files.

The YAML code listed in Figure 4 shows one of several sets of default properties that can be mixed in to a generated program. The properties show are managed using a graphical interface program before injection into the StringTemplates which perform the C++ code emission. These parameters correspond to the model parameters in the right hand column of Ta-



[✓ DEBUG] Optimise Compactify

Title:	An SPG Program
Name:	mysimprog
Author:	A.N. Onymous
Start Date:	1.8.2011
Makefile	Makefile
main() file	mysimprog.cpp

Complex Field Simulation Program
Uses QFT Model
Compatible with Cartan

See CSTN-123

	ID	type	hyphenName	defaultValue	commentDescriptor
0	temperature	double	T	2.269	temperature
1	probability	double	p	0.5	probability
2	lengthSpecifier	string	L	16 16	gths and implied dimension
3	nCells	int	N	64	number of cells in system
4	nRuns	int	runs	1	endently seeded runs to do
5	nSteps	int	steps	100	r of steps to evolve model
6	equilibriumSteps	int	equil	0	libration steps to discard
7	seed	long long int	seed	0	rng seed
8	doTimer	bool	time	false	switch on program timer
9	doConfigurationDumping	bool	dump	false	dump configurations
10	doMeasuring	bool	measure	false	switch on measurements
11	jobDescriptor	string	job		job descriptor prefix
12	nStates	int	Q	2	number of states

Figure 3: SPG Screen-dump - command line argument table view showing edit-able program parameters, and buttons and switches to control support analysis tools within the Java GUI controller.

ble 1. They have specific types that vary from model to model. Some need to be compiled in, and others need generated code to support run-time choices - using the job controlling shell script for example.

The GUI code is written in Java and essentially just manages a set of forms consisting of Buttons, Switches, edit-able TextFields and other property manipulation widgets. Although there are other and lighter-weight software tools available for managing forms, the use of Java means that a rich set of additional support and checking facilities can be added into the GUI relatively easily. Some of the features available from the table view shown in Figure 3 include: type checking fields; filling in missing default values; editing augmenting comment descriptors; importing (and hence mixing in) multiple archived parameters from previous work; controlling the level of debug information generated or the level of optimisation information to be included. The GUI program then can invoke the StringTemplate generator apparatus to fill in the various C++ code specification template files and group template functions, injecting the various user specified attributes and properties for the particular model in question.

The ST program template code listed in Figure 5 shows the StringTemplate used to generate a main() program for our standard family of simulation programs. The template file specifies varies code sections which are filled in using template functions help in a StringTemplate group file. The details are injected into the templates through Java HashMap Objects which are constructed to contain key-value pairs for the various attributes and properties needed for each simulation model. StringTemplate supports a number of useful functional language constructs such as list comprehension and template function application that allows us to locate all the code emission settings in the template and restrict the Java graphical interface code to management of the properties separately.

```
# Yaml-formatted Default Command Line Args
-
  ID: temperature
  type: double
  hyphenName: T
  defaultValue: 2.269
  commentDescriptor: "temperature"
-
  ID: probability
  type: double
  hyphenName: p
  defaultValue: 0.5
  commentDescriptor: "probability"
-
  ID: lengthSpecifier
  type: string
  hyphenName: L
  defaultValue: "16␣16"
  commentDescriptor: "lattice␣lengths"
```

Figure 4: Yaml file containing default argv options information for a SPG-generated program. The properties are used to specify generation of a simulation program as in Figure 1.

```
$COMMENT_HEADER()$
$HASH_INCLUDES()$
$COMMENT_LINE_132()$
int main( int argc, char *argv[] ){
  ProgramTimer timer; // library object
  $GLOBALS_DECL(declList)$
  $ARGV_DECL(argvList)$
  $ARGV_SWITCH_STATEMENT(argList)$
  $INIT_STATEMENTS(initList)$
  $LOOP_STATEMENTS(loopList)$
  $STATISTICS_STATEMENTS(statsList)$
}
$COMMENT_FOOTER()$
```

Figure 5: ST simulation program template, invoking various template functions.

The StringTemplate ability to separate out template functions from the file template means that boilerplate code can be easily written or cut-and-pasted from known working program source files. The model-driven details that vary from different simulation projects are then uncluttered and easier to develop free of the boilerplate code. This skeletal approach is particularly powerful to augment the many cases of complex interdependencies that mean a simple library routine or library object approach ins not possible.

Model Name	Crafted C++ LOC	Generated SPG/C++ LOC
Ising	1,678	180 / 45
Q-State Potts	1,738	180 / 85
Classical XY (Clock)	2,112	222 / 345
Classical Heisenberg	2,220	222 / 398
Eden growth	548	130 / 34
Spatial Epidemic	670	130 / 46
Wolfram Automata	210	112 / 22
Game of Life	245	112 / 29
N_S species RPSLS	970	310 / 107
Predator-Prey	1,300	310 / 368
Lotka-Volterra	1,430	340 / 397
Cahn-Hilliard	800	245 / 76
TDGL	880	245 / 88

Table 2: Some line-of-code(LOC) estimates of original hand-crafted codes with a combination of the simulation code generator specification code and manually added C++.

The SPG tool developed and discussed for this paper can generate partially completed programs as well as fully generated ones. Sometimes is is sufficient to generate just a subroutine or lengthy case statements where the generator does not make typing mistakes. These partial code fragments can still help the application developer considerably, even if they are themselves then just cut-and-pasted into a final source code.

5 Results

A useful metric to assess the value of this code generation approach to simulation programs is to compare the (approximate) code size of hand-crafted simulation programs with that of the auto-generated equivalents. This is not completely simple as in all cases it is very difficult to make the prototype code generator produce a complete product. It was designed as an aid to the developer not a replacement so we should compare the combination of lines of specification or template code and the additional C++ lines added by hand to the lines of source in the original fully hand written programs.

Table 2 shows some properties of the generated simulation codes alongwith comparisons against hand-crafted application codes. The data were obtained from simple Unix utility "wc" word-count sums on the source code files involved. Figures for lines of code are inevitably estimates based on judgments on what parts to count or not however. Figures give for hand generated C++ LOC include measurements and statistical analysis code. Figures given for the semi-automatically generated codes consist of the lines of template code needed, as well as the lines of additional C++ custom code that was required to fill in the skeletons. The table shows a definite trend of a low ratio of additional custom C++ LOC to specifier template LOC where the template had been well thought through and was a good fit to the model in question.

6 Discussion

Although the hand-crafted codes have been developed over a 20 year period, the data still represent reasonably consistent estimates across the model applications reported. Typically a substantial reduction in amount of total LOC required was achieved for all the models. For models where they "matched" the "pattern" of the developed template the reduction in LOC was around a factor of ten, but in all cases was at least a factor of two.

It is difficult to argue that the code complexity has been reduced by these factors, as arguably some of the template function code is considerably more complex and would be more challenging to an application domain developer than would the plain ordinary C++ boilerplate code. It is not a simple comparison to make. Nevertheless, in the longer term these figures do represent quite a dramatic reduction in some code-complexity maintainability metric. It is also important to realise that the template code is shared across the whole family of programs described in table 2 and this only has to be maintained as a single set of templates and not as duplicated boilerplate code across several C++ files.

The prototype tool and approach adopted has been very much geared towards assisting the application developer in their work-flow and not replacing a whole stage in the developer pipeline. In that sense it is an open question at what point the generated code must be maintained as C++ code and at what point it is still automatically re-generatable from templates. If an idea is propagated back into the templates it is not easy to regenerate the skeletons without losing or damaging the custom code that was added by hand. The challenge is still to find a code structural mechanism so that new skeletons could be subsequently regenerated in place of old ones, with such damage. The use of code regions and region sub-templates may be a possible approach to this problem.

Our prototype system has served primarily as a platform for exploring DSL/ACG technologies and tools. A design methodology that records and structures features such as a feature description language [49, 50] and tool-set holds some promise for managing the properties of a planned simulation program for automatic or semi-automatic generation.

7 Conclusion

We have described some ideas in domain specific languages and automatic code generation as they apply to an interactive prototype tool for assisting in generating fast C++ simulation programs of complex model systems. We have described how Parr's StringTemplate tool combined with Java Graphical Interface Libraries and YAML parsers have been combined to produce the "SPG" tool. We have presented this primarily as an exploration into model-driven software engineering techniques for automatic code generation. However we have also shown through some preliminary and approximate estimates of lines of code (C++ or combined template code with added C++) that some dramatic reductions in boilerplate lines of code are possible. This obviously has interesting implications for longer r term development of application specific codes and associated support tools.

In this article we focused on lattice-based simulation models, albeit with a range of different cell types ranging from bits to vectors of complex number. There are however other application areas for future work such as graph based structure models like damaged systems or small-world systems where these techniques may also be applicable. Other areas include generating different target languages such as parallel computing language source to make use of different hardware platforms and incorporating appropriate numerical libraries.

In conclusion, this is a very promising approach with impressive medium to long-term advantages, albeit with a steep learning curve to integrate the tools together in the short-term. Managing the types of information in the various parts of the tool has been the greatest technical challenge. This model-driven ACG approach does however seem likely to make a considerable impact in speeding up various investigations and numerical experiments across various disciplines in computational science.

References

[1] Denning, P.J., Riehle, R.D.: Is software engineering engineering? Communications of the ACM **52** (2009) 24–26

[2] Glass, R., Vessey, I., Ramesh, V.: Research in software engineering: an analysis of the literature. Information and Software Technology **44** (2002) 491–506

[3] Sangwan, R.S., Lin, L.P., Neill, C.J.: Structural complexity in architecture-centric software evolution. Computer **41** (2008) 96–99

[4] Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving extensibility through product-lines and domain-specific languages: A case study. ACM Trans. Software Engineering and Methodology **11** (2002) 191–214

[5] Consel, C.: From a program family to a domain-specific language. In: Domain-Specific Program Generation. (2003) 19–29

[6] Voelter, M., Visser, E.: Product line engineering using domain-specific languages. In: Proc. 15th Int. Software Product Line Conference (SPLC 2011), Munich, Germany (2011)

[7] Spinellis, D.: Notable design patterns for domain-specific languages. Journal of Systems and Software **56** (2001) 91–99

[8] Ward, M.: Language oriented programming. Software - Concepts and Tools **15** (1994) 147–161

[9] van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Notices **35** (2000) 26–36

[10] Fowler, M.: Domain-Specific Languages. Number ISBN 0-321-71294-3. Addison Wesley (2011)

[11] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys **37** (2005) 316–344

[12] Lengauer, C., Batory, D., Consel, C., Odersky, M., eds.: Domain-Specific Program Generation. Number 3016 in LNCS. Springer (2003) ISBN 3-540-22119-0.

[13] Hawick, K., Leist, A., Playne, D.: Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs. Int. J. Parallel Prog. **39** (2011) 183–201

[14] Hawick, K.A., Johnson, M.G.B.: Bit-packed damaged lattice potts model simulations with cuda and gpus. Technical Report CSTN-134, Computer Science, Massey University, Albany, Auckland, New Zealand (2011)

[15] Kim, B.J., Hong, H., Holme, P., Jeon, G.S., Minnhagen, P., Choi, M.Y.: Xy model in small-world networks. Phys. Rev. E **64** (2001) 056135

[16] Anderson, P.W.: New approach to the theory of superexchange interactions. Phys. Rev. **115** (1959) 2–13

[17] Eden, M.: A two-dimensional growth process. In: Proc. Fourth Berkeley Symposium on Mathematics, Statistics and Probability. Volume 4., Berkeley, Univ. California Press (1960) 223–239

[18] Agiza, H.N., Elgazzar, A.S., Youssef, S.A.: Phase transitions in some epidemic models defined on small-world networks. Int. Journal of Modern Physics C **14** (2003) 825–833

[19] Wolfram, S.: Cellular Automata as models of complexity. Nature **311** (1984) 419–424

[20] Gardner, M.: Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life". Scientific American **223** (1970) 120–123

[21] Hawick, K.: Complex Domain Layering in Even Odd Cyclic State Rock-Paper-Scissors Game Simulations. In: Proc. IASTED International Conference on Modelling and Simulation (MS2011). Number CSTN-066, Calgary, Alberta, Canada (2011) 735–062–1–8

[22] Hawick, K.A., Scogings, C.J., James, H.A.: Defensive spiral emergence in a predator-prey model. Complexity International (2008) 1–10 ISSN ISSN 1320-0682.

[23] Hawick, K.A.: Spectral analysis of growth in spatial lotka-volterra models. In: Proc. IASTED International Conference on Modelling and Simulation. Number CSTN-092, Gabarone, Botswana (2010)

[24] Hawick, K.A., Playne, D.P.: Modelling, Simulating and Visualizing the Cahn-Hilliard-Cook Field Equation. International Journal of Computer Aided Engineering and Technology (IJCAET) **2** (2010) 78–93

[25] Hawick, K.A., Playne, D.P.: Numerical Simulation of the Complex Ginzburg-Landau Equation on GPUs with CUDA. In: Proc. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN). Number CSTN-070, Innsbruck, Austria, IASTED (2011) 39–45

[26] Cong, J.: Overview of center for domain-specific computing. Journal of Computer Science and Technology **26** (2011) 632–635

[27] Czarnecki, K., O'Donnell, J.T., Striegnitz, J., Taha, W.: Dsl implementation in metaocaml, template haskell, and c++. In: Domain-Specific Program Generation. (2003) 51–72

[28] Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: Proc. ACM PLDI'11, San Jose, California (2011) 132–141

[29] Sadilek, D.A.: Prototyping and simulating domain-specific languages for wireless sensor networks. Technical report, Humboldt-Universitat zu Berlin, Institute for Computer Science (2007)

[30] Consel, C., Réveillère, L.: A dsl paradigm for domains of services: A study of communication services. In: Domain-Specific Program Generation. (2003) 165–179

[31] Hammond, K., Michaelson, G.: The design of hume: A high-level language for the real-time embedded systems domain. In: Domain-Specific Program Generation. (2003) 127–142

[32] O'Donnell, J.T.: Embedding a hardware description language in template haskell. In: Domain-Specific Program Generation. (2003) 143–164

[33] Cong, J., Sarkar, V., Reinman, G., Bui, A.: Customizable domain-specific computing. IEEE Design & Test of Computers **March/April** (2011) 6–14

[34] Kennedy, K., Broom, B., Cooper, K., Dongarra, J., Fowler, R., Gannon, D., Johnsson, L., Mellor-crummey, J., Torczon, L.: Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. Journal of Parallel and Distributed Computing **61** (2001) 1803–1826

[35] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., J. Turian, D.W.F., Bengio, Y.: Theano: A cpu and gpu math expression compiler. In: Proceedings of the Python for Scientific Computing Conference (SciPy) 2010, Austin, TX, USA. (2010)

[36] Logg, A., Wells, G.N.: Dolfin: Automated finite element computing. ACM Trans. Math. Soft. **37** (2010) 1–28

[37] Hawick, K.A., Playne, D.P.: Automatically Generating Efficient Simulation Codes on GPUs from Partial Differential Equations. Technical Report CSTN-087, Computer Science, Massey University (2010) Submitted to Elsevier, J. Comp. Sci.

[38] Terrel, A.R.: From equations to code: Automated scientific computing. Computing in Science & Engineering **March/April** (2011) 78–82

[39] Lengauer, C.: Program optimization in the domain of high-performance parallelism. In: Domain-Specific Program Generation. (2003) 73–91

[40] Hudak, P.: Modular domain specific languages and tools. In: in Proceedings of Fifth International Conference on Software Reuse, IEEE Computer Society Press (1998) 134–142

[41] Ghosh, D.: Dsl for the uninitiated - domain-specific languages bridge the semantic gap in programming. Communications of the ACM **54** (2011) 44–50

[42] Beckmann, O., Houghton, A., Mellor, M.R., Kelly, P.H.J.: Runtime code generation in c++ as a foundation for domain-specific optimisation. In: Domain-Specific Program Generation. (2003) 291–306

[43] Vargas, J., Duarte, H.: 3p. a first approach to a domain specific language for constructing code generation. In: Proc. Int Conf. on Information Resources management (CONF-IRM 2010). (2010) 1–16 Paper 15.

[44] Olgaard, K., Wells, G.: Optimisations for quadrature representations of finite element tensors through automated code generation. ACM Trans. Math. Software **37** (2010) 8:1–8:23

[45] Kosar, T., Oliveira, N., Mernik, M., Pereira, M.J.V., matej Crepinsek, da Cruz, D., Henriques, P.R.: Comparing general-purpose and domain-specific languages: An empirical study. Computer Science and Information Systems **7** (2010) 247–264

[46] Parr, T.: Language Implementation Patterns, Create Your Own Domain-Specific and General Programming Languages. Number ISBN 978-1-93435-645-6. Pragmatic (2010)

[47] Parr, T.: The Definitive ANTLR Reference - Building Domain-Specific Languages. Number ISBN 978-0-9787392-5-6. Pragmatic Bookshelf (2007)

[48] Parr, T.: A functional language for generating structured text. Technical report, University of San Francisco (2006)

[49] van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. J. Computing and Info. Tech. **10** (2002) 1–17

[50] Schobbens, P.Y., Trigaux, P.H.J.C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks **51** (2007) 456–479