Simulation Software Generation using a Domain-Specific Language for Partial Differential Field Equations

K.A. Hawick and D. P. Playne
Computer Science, Massey University, North Shore 102-904, Auckland, New Zealand
email: \{k.a.hawick, d.p.playne\}@massey.ac.nz
Tel: +64 9 414 0800  Fax: +64 9 441 8181
April 2013

ABSTRACT

Domain-specific language techniques can considerably lower the software development effort and time required for problems in computational science and engineering. We describe our domain specific language for field-based partial differential equation simulations and show how it can address a whole family of such problems. Our system requires minimal effort to generate C++ software for a new equation model, but also dramatically lowers the effort needed to generate code in a different output language. We report on the lines of code for several example problems discuss software engineering implications of this automatic code generation approach.

KEY WORDS

generative programming; DSL; partial differential equation.

1 Introduction

Many problems in computational science and engineering can be formulated in terms of partial differential equations (PDEs). A common simulation pattern involves the time-integration of an initial value model where the system is defined on a spatial mesh with spatial calculus operators in the equation. Although the mathematical and numerical methods for solving such problems are well known, it is still a tedious and error-prone task to write correct and efficient software for a new problem.

Although a great deal of techniques [1] are known for building optimising compilers [12, 15, 30] the goal of automatic parallelising compilation remains elusive. Some important progress was made for some data-parallel constructs [5, 29]. However it seems likely that there are some general problems that compiler generators will probably never be able to solve completely, without programmer assistance [32]. More optimistically however it is feasible to look at some specific classes of application domain problems and use application domain-specific languages and tools to address them.

In this paper we report on how modern compiler-level tools and technology can be used to make a generative programming system that can create fast and readable data-parallel software for solving some PDE based problems. Four example PDEs that fit into the example category are introduced - the Heat, Cahn-Hilliard, Ginzburg-Landau and Spatial Lotka-Volterra equations. These PDEs are first-order in the time derivative but contain second- or fourth-order spatial calculus operators. Example illustrations of the Cahn-Hilliard, Ginzburg-Landau and Lotka-Volterra equations are shown in Figure 1. We explain how a plain ASCII expression of these mathematical equations can be parsed and used to generate software in a language like C or C++. This is our domain-specific language for formulating field-based PDE applications.

We show how a relatively minor change to the mathematical specification of the equation allows a whole new code to be generated relatively trivially. These approaches combined in saving on: programmer time; testing effort; and production run time. This system supports the investigation of whole families of problems that would hitherto have taken a lot longer to tackle.

A number of software systems and algebraic problem solving environments allow users to automatically generate solver source code in standard programming languages such as Fortran [10, 11, 23]. A number of systems also address the problem of generating parallel code [35]. Research projects [3, 24] and commercial problem solving systems such as Matlab [31] or Mathematica [34] also support code generation from a mathematical formulation of equations. We are interested
Although there are a number of mathematical and numerical approaches such as finite-elements that can be expressed using this approach to code generation, we focus in this paper on regular mesh problems that can be solved using finite-difference methods. We defer a detailed discussion on different numerical time integration techniques and different stencil operators for the spatial calculus to another work [28].

The idea of generating PDE solver software is not new. As long ago as 1970, Cardenas and Karplus experimented with manually written programs that combine both translation and generation in a single ad hoc stage [8] partial differential equation language (PDEL) based on PL/1 syntax. Some important work is being done by Logg and collaborators on the semi-automatic generation of Finite Element algorithms. The FENICS [20] and DOLFIN [21] projects take a somewhat different approach to the one we do, making more heavy use of linear algebraic methods and the associated separately-optimised software for solving linear algebra and matrix-oriented problems such as BLAS [13], BLACS [14], LAPACK [4] and ScaLAPACK [9]. While it is also possible to formulate the Finite Difference methods that we employ using full matrix methods too, we focus here on direct methods and formulations for regular meshes that do not need full matrices and that make use of explicit sparse data storage methods. This allows us the luxury of worrying less about storage space for the spatial calculus and thus being able to experiment more readily with higher-order time integration methods which themselves require multiple copies of the field data for intermediate fractional time steps.

In this article we discuss the general form of applicable partial differential field equation problems in Section 2. The structure and operation of our parser and code generator is given in Section 3. We present some generated code examples and associated run-time performance data in Section 6 and discuss associated issues in Section 7. We offer some conclusions in and ideas for future work in Section 8.

## 2 Solving Field PDEs

Many interesting problems in physics, chemistry, biology and other areas of science can be formulated as partial differential field equations that evolve over time. These formulations fall into the general pattern:

$$\frac{du(r,t)}{dt} = F(u,r,t)$$  \hspace{1cm} (1)

where the time dependence is first order and the spatial dependence in the right hand side is often in terms of partial spatial derivatives such as \(\nabla_x, \nabla_y, \nabla_z, \nabla^2, \nabla^2 \cdot \nabla^2, \ldots\). Some well-known problems that fit this pattern are:

The **Heat equation** which models how heat is distributed through a material over time. The heat distribution can be defined in terms of the scalar field \(u\) and \(\alpha\) which is a positive constant representing the thermal diffusivity.

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$  \hspace{1cm} (2)

The **Cahn-Hilliard equation** [7, 17] which models a quenching binary alloy and is expressed in terms of a scalar field \(u\):

$$\frac{\partial u}{\partial t} = m \nabla^2 \left(-bu + Uu^3 - K \nabla^2 u \right)$$  \hspace{1cm} (3)

where it is usual to truncate the series in the free energy [6] at the \(u^4\) term, although some work has used up to the \(u^6\) term [33].

The **Time-Dependent Ginzburg Landau equation** [22] which can be used to describe the macroscopic behavior of superconductors can be defined in terms of a complex scalar field \(u\):

$$\frac{\partial u}{\partial t} = \frac{b}{i} \nabla^2 u - \frac{q}{i} |u|^2 u + \gamma u$$  \hspace{1cm} (4)

The **Lotka-Volterra** equation is often written as:

$$\frac{dP}{dt} = F(P)$$  \hspace{1cm} (5)

where \(P\) might be vector of several population variables for predator and prey species and \(F\) might incorporate a matrix of cross-coupling terms and spatial calculus operators. This equation can be formulated as a two-species predator-prey model using the Laplacian operator for spatial coupling [16]. This can be defined as:

$$\frac{du_0}{dt} = D_0 \nabla^2 u_0 + Au_0 - Bu_0 u_1$$

$$\frac{du_1}{dt} = D_1 \nabla^2 u_1 + Du_0 u_1 - Cu_1$$  \hspace{1cm} (6)

where \(u_0\) is the prey population and \(u_1\) is the predator population.

In some cases the full details of the right-hand sides of these sort of equations are known and are immutable parts of the field model. In other cases a family of equations can be generated by using different expansions or approximates. A good example is the Cahn-Hilliard equation where the free-energy term is usually approximated by a polynomial with second and fourth order terms, but alternatives such as including higher order terms make sense but are hard (tedious and error-prone) to implement.

A powerful idea to address implementation difficulties is therefore software tool that can help generate lines of code in a standard programming language like C, C++, D, Java, Fortran, that implements one of the standard numerical approaches to solving the equation in question. There are some well known lines of approach to solving the numerical integration in time - storing the state of the entire model field that expresses the right hand side and applying second order methods...
such as the midpont method (aka second-order Runge-Kutta) or higher-order methods such as the well-known Runge-Kutta Fourth-order method as appropriate.

3 Parser and Generator Structure

In this section we describe the various components of our generative programming prototype - known as “Simulation Targeted Automatic Reconfigurable Generator of Abstract Tree Equations (STARGATES).” This system assembles simulation code of partial-differential field equation(s) from different simulation components. It is important to make the distinction between the model, the simulation and the implementation. The model is the equation(s) and the parameters of the model. The simulation is the specific combination of a model and the methods used to simulate it - the lattices, spatial stencils, boundary conditions and numerical methods. The implementation is the target specific code that can be compiled into machine code to compute that simulation.

The system will take the different components of the simulation and use them to construct an abstract “simulation” object that contains all the relevant information. To construct this object all the components must be combined together in an appropriate way. The stencils used by the equation must be supplied and matched with the type of lattice the simulation is using. The integration method must be combined with the equation to form the calculations of each step of the integration. The final combination of components forms the simulation object.

This object can be then queried by an output generator to produce code that performs the simulation using a desired target programming language. Some additional configuration information about the simulation must also be supplied to define properties of the simulation such as system size, parameter values etc. Different output generators will produce different code which represents the different possible implementations of the same simulation. The architectural structure of the generative system prototype is shown in Figure 2.

![Figure 2: Structure and logical flow of STARGATES.](image)

The system allows the user to write the equation(s) that define the model in a mathematical form using ASCII. The Equation Parser reads this ASCII and constructs a tree representing the equation. The equation tree will contain all the information required by the rest of the system to generate output code for that simulation. Parsing mathematical equations is a potentially open ended problem but as indicated we are able - for our prototype tool - to restrict the equation forms we are addressing to some specific patterns, and make the problem tractable.

To write the Equation Parser, we have made use of the compiler generator technology ANTLR [25]. ANTLR is a relatively modern tool building upon historical developments [2] including the well known lexing/parsing tools: lex/yacc [19] and flex/bison [18, 26]. ANTLR allows us to specify a simple grammar from which ANTLR will automatically generate a Lexer and a Parser. The grammar shown here supports the declaration of parameters, lattices and equations with simple mathematical operators +, -, * ,/. The advantage of using ANTLR is that it is very easy to extend and change the grammar as necessary. A simple version of the grammar is shown in Listing 1.

```plaintext
Listing 1: Simple equation ANTLR grammar.

DIGIT : '0' . . '9';
CHAR : 'a' . . 'z' | 'A' . . 'Z' | ' ';
ID : CHAR (CHAR|DIGIT)*;
NUM : (DIGIT)+ ('.' (DIGIT)+ )?;
DERIVATIVE : 'd / d t ' ;
FUNC : ( 'ABS' | 'SQRT' ) ;
...

file : (statement) + EOF ! ;
statement : ( declaration | equation ) ;
declaration : ID ':' ID ;
equation : DERIVATIVE ID '=' additive ';';
additive : mult ('+' | '-' ) mult * ;
multi : unary ( '*' | '/' ) unary * ;
unary : atom
    | MINUS atom ;
atom : NUM
    | ID
    | '(' additive ')' ;
    | ID '{' additive '}' ;
```

This grammar is sufficient to parse equations such as the Cahn-Hilliard model (see equation 3) in the form:

```plaintext
floating M;
floating B;
```
where the “equation” start-point defines a first order time-differential equation, whose right hand side has a number of spatial calculus operators as well as algebraic combinations of the fundamental field and parameters.

This equation is processed by the ANTLR generated parser which will construct a tree representing the model. This tree includes the types, parameters, fields and equations of the model. For example the tree created for the Cahn-Hilliard equation (See equation 3) is shown in Figure 3.

After the initial equation is parsed, the tokens are converted into a tree which can then be parsed by a ANTLR tree parser. This tree parser constructs a tree representing the equation out of objects that are each equivalent to a component of the equation (parameters, operators, stencils etc). Given the example of the Cahn-Hilliard equation (See equation 3), when this equation is parsed, the ANTLR tree parser generates the tree shown in Figure 3.

![Figure 3: Two equation trees representing the Cahn-Hilliard Equation. The initial tree created by the ANTLR parser (left) and the tree after it has been rearranged by the Stencil Library to contain no nested stencil nodes (right).](image)

The tree contains all of the information about the equation needed by the system. To form a simulation objection, this equation tree must be combined with an integration method. However, before this is done, the stencils used by the equation must first be resolved. The Stencil Library provides the necessary stencils and in some cases may manipulate the equation tree. This is best performed before the tree is combined with an integration method.

This equation parser is capable of processing equation files that contain multiple fields and multiple equations. This is necessary for equations such as the Lotka-Volterra model which has a field for each of the multiple species populations and equations governing their interactions. The spatial predator-prey form of the Lotka-Volterra model can be described in ASCII form as follows:

```plaintext
float A;
float B;
float C;
float D;
float D0;
float D1;
float[] u0;
float[] u1;
d/dt u0 = A*u0 - B*u0*u1 + D0 * Laplace{u0};
d/dt u1 = -C*u1 + D*u0*u1 + D1 * Laplace{u1};
```

5 Output Generator

The output generators are responsible for querying the simulation object and creating language-specific output code. These generators glean the information they need from the simulation object and combine that information with language specific code templates to produce an implementation of the simulation. There is very little restriction placed by the system on output generators. Multiple generators can be created to target the same language but use different simulation structures or alternatively one generator can have many configuration options to produce simulations with different structures. Because the generators have access to the context of the simulation, they can introduce specific optimisations when appropriate.

Code generators can be constructed for many different sequential and parallel programming languages [27]. The generators will differ in terms of the instructions they produce and the general code structure. The syntax of the generators will depend on the target language, but generators based on the same syntax will often share similar components. For example, both the C++ and CUDA generator use C-style syntax so several elements of the target code will be the same. The high-level structure of the code will be dependent on the type of implementation they are producing. The example code structure template for a C++ implementation is shown in Algorithms 1.
The generators also have many operation templates which are populated with data from the simulation object to perform operations required by the generator. These operations include allocating memory for a lattice, initialising parameters, calling functions etcetera. Some examples of the instructions produced by these output generators are shown in Section 6.

The advantage of this approach is that the front-end parsing and simulation object construction for the simulations remains the same regardless of the output generator used. When a new architecture or language is released, a new generator can be written that will allow all existing simulations to be migrated to make use of that new architecture or language. This makes it much easier to adopt a new language or architecture without the need to rewrite the entire simulation base. This is a far easier and more extensible programming model than maintaining separate code versions for each simulation and computing architecture.

6 Results

The system currently has a number of output code generators that can produce target code of simulations for a number of computing architectures. The main code generator discussed in this work is for single-threaded C++ code generation. However, the system can also generate code for multi-core CPUs using TBB or pThreads, for distributed machines using MPI and for Tesla, Fermi or Kepler generation GPU devices using CUDA.

Listing 2: Generated code sample produced by the C++ generator.

```c
int main() {
    float *u, *u0, *u1, *u2;
    u = new float[Y * X];
    ...
    for (int t = 0; t < 1024; t++) {
        rk2(u0, u1, u2, h);
        swap(u0, u2);
        memcpy(u, u0, Y * X * sizeof(float));
    }
    void rk2(float *u0, float *u1, float *u2, float h) {
        for (int iy = 0; iy < Y; iy++) {
            for (int ix = 0; ix < X; ix++) {
                ...
            }
        }
    }
}
```

Here we present the code generated by the two output generators for the Cahn-Hilliard equation (see Equation 3). One of the generators builds a single-threaded C++ program and the other generates a CUDA program optimised for Fermi architecture GPUs. Sample generated code can be seen in Listing 2 which shows the general structure of main function and integration method for a C++ simulation of the Cahn-Hilliard model using the Runge-Kutta 2\textsuperscript{nd} order integration method. The generator creates and initialises the main mesh of the equation \( u \). It also creates the three meshes required by the RK2 method \( u0, u1 \) and \( u2 \). Also shown in Listing 2 is the function to perform the integration steps, in this code both of the RK2 steps are performed in one C++ function.

Since both the C++ and CUDA generator stages use C-like syntax, the code to perform the actual equation is the same for both generators. This code (with whitespace formatted for ease of reading) is shown in Listing 3. This code calculates the change in one spatial cell for the Cahn-Hilliard equation \( u(x, y) \). We have tried to make the variable names and code layout closer to human readable choices than some code generators do since the programmer may decide to adopt the generated code and include it in a code package that is subsequently human-maintained rather than regenerated.

Listing 3: The same equation calculation code generated by both the C++ and CUDA generators.

```c
M*(u_ymlx + u_yxml1 + (-4*u_yx) + u_yxpl + u_yplx) +
U*(u_ymlx*u_ymlx*u_ymlx) + (u_yxml1*u_yxml1*u_yxml1) +
(-4*u_yx*u_yx*u_yx*u_yx) +
(u_yxpl*u_yxpl*u_yxpl) +
(u_yplx*u_yplx*u_yplx)) -
K*( (2*u_ymlxml) + (-8*u_yxml) + (2*u_yxpl) +
    u_yx2m + (-8*u_yxmlml) + (20*u_yx) +
    (-8*u_yxpl) +
    u_yx2) +
    (2*u_yplxml) + (-8*u_yplx) + (2*u_yplxpl) +
    u_ypl2x)
```

One of the major advantages of a code generator is the reduced effort required to produce an implementation of a simulation. Programmer effort is difficult to measure but the number of lines of code required to define a simulation can be used as an approximation. To define a simulation the programmer must define both the model (fields, parameters and equations) and the configuration (parameter values, integration method, lattice geometry etcetera). Table 1 shows the number of lines of code defined by the programmer (definition) and the number of generated lines of code for a number of different simulations.

The code that the generator produces obviously does not contain every possible optimisation as humans are usually much better at identifying which optimisations are applicable for particular simulations. However, if the pattern of possible optimisation is identified, it could subsequently be incorporated into an output generator. The generator can identify optimisations that cannot easily be found by compilers due to the higher level of information available to the generator.
We have deferred discussion of code generation for parallel platforms to [28]. There is scope for applying the approach we have outlined to many other members of this class of time-integrated partial differential equation field equations.

### 8 Conclusions and Future Work

In summary, we have described how a staged parser and tree-walking code generator can produce device agnostic software for modern platforms, where the software is optimized, human-readable and maintainable. This is possible as we have focused on a very specific form of application domain problem - solving regular partial differential equations using finite difference equations. The speed performance of the generated code is very close to that attainable by expert hand-generated software but with considerably less time required to develop and test a new equation or indeed to deploy for a new platform.

One important outcome of this work for us is the ability to investigate whole families of problems rather than having to focus on just one hand-coded one. Problems like the Cahn-Hilliard equation or the Time-Dependent Ginzburg-Landau equation have a number of choices embedded in them that, while compactly expressible in mathematics, lead to quite different software formulations. A tool like this opens up a number of feasible investigations in computational physics that would otherwise be quite time consuming - and in the past have consumed a whole PhD-worth of research effort each in terms of coding, testing and general experimental effort.

A more general outcome of this work is the software architecture for scientific problem domain specific languages that can be parsed and can have output code generated in a number of different target languages and associated platforms. We note the promise of modern compiler generator tools such as ANTLR and the benefits of using them rather than attempting a monolithic single stage parser-generator tool.

The domain-specific language approach is a powerful one for lowering the software engineering effort required for investigating problems in computational science. There is considerable scope for expanding the simulation model-driven approach we have taken to other problems and platforms.
References