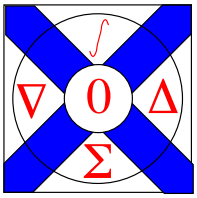

Computational Science Technical Note **CSTN-166**

Fluent Interfaces and Domain-Specific Languages for Graph Generation and Network Analysis Calculations

K. A. Hawick

2013

Internal Domain-Specific Languages (DSL) provide an elegant mechanism to reduce the code complexity of complex systems simulations programming. Many complex systems problems manifest themselves as networks. Graph analysis techniques can be employed to count the number of: separate components; circuits; paths; or distances in the network. Several other properties such as: mean and peak degree; component or community size; adjacency eigen-spectra, and so forth can also be calculated. Calculation of these properties gives a signature that can help classify a network as belonging to a particular category with known behaviours. In practice however, importing applications data into graph analysis software and managing these calculations can be complex. Domain-specific language techniques allow a high-level graph calculations language to be developed that invokes software components in a graph manipulations framework. We describe a prototype graph generation and analysis domain-specific language built using fluent interface techniques and the Java programming language. We report on: attainable code complexity reduction, framework computational performance, and software engineering directions for internals DSLs for this sort of applications problem.

Keywords: domain-specific language; fluent interface; graph generation; graph computations; programming tools and languages

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-166,
        author = {K. A. Hawick},
        title = {Fluent Interfaces and Domain-Specific Languages for Graph Generation
                and Network Analysis Calculations},
        booktitle = {Proc. Int. Conf. on Software Engineering (SE'13)},
        year = {2013},
        pages = {752-759},
        address = {Innsbruck, Austria},
        month = {11-13 February},
        publisher = {IASTED},
        institution = {Computer Science, Massey University},
        keywords = {domain-specific language; fluent interface; graph generation; graph
                computations; programming tools and languages},
        owner = {kahawick},
        timestamp = {2012.12.01}
}
```


Prof Ken Hawick, Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand.

Technical Report CSTN-166

Fluent Interfaces to a Java-Based Internal Domain-Specific Languages for Graph Generation and Analysis

K.A. Hawick
Computer Science, 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

October 2012

ABSTRACT

Internal Domain-Specific Languages (DSL) provide an elegant mechanism to reduce the code complexity of complex systems simulations programming. Many complex systems problems manifest themselves as networks. Graph analysis techniques can be employed to count the number of: separate components; circuits; paths; or distances in the network. Several other properties such as: mean and peak degree; component or community size; adjacency eigenspectra, and so forth can also be calculated. Calculation of these properties gives a signature that can help classify a network as belonging to a particular category with known behaviours. In practice however, importing applications data into graph analysis software and managing these calculations can be complex. Domain-specific language techniques allow a high-level graph calculations language to be developed that invokes software components in a graph manipulations framework. We describe a prototype graph generation and analysis domain-specific language built using fluent interface techniques and the Java programming language. We report on: attainable code complexity reduction, framework computational performance, and software engineering directions for internals DSLs for this sort of applications problem.

KEY WORDS
domain-specific language; fluent interface; graph generation; graph computations; programming tools and languages.

1 Introduction

Engineering elegant and understandable software for simulating complex systems is a challenging problem. Complex models based on graphs and network structures present an opportunity to abstract over some of the common data structures and traversal algorithms using a domain-specific programming language.

Graph and network problems arise in many different application domains, including: physical infra-structural systems such as electricity, water and power networks; social, financial and market systems; overlays such as web-page relationships on the Internet; and physical systems in materials science. Analysing such systems is not necessarily simple as they come in different formats. A useful approach is to be able to compare different individual network sets with families or common classes of graph pattern to try to match up characteristic signature properties. To this end a domain-specific language for graph calculations is a desirable tool to ease the management of graph and network calculation experiments [13].

The designers of computer programming languages try to help the programmers express their ideas and algorithms concisely and clearly. However even with the most elegant programming language difficulties arise when programs become significantly bigger and more complex. To an extent the standard computer science approach of "divide and conquer" can be applied to abstractify the relevant ideas and components of a large program into software libraries that can be developed, tested and hidden away, to be invoked only when needed. This lowers the amount of program code the programmer needs to display on screen or in the mind at once and is one of the most important aspects of managing the development of large-scale complex software [7].

Although many modern programming languages have introduced various feature to further this abstraction and the consequent lowering of code complexity applications developers are still easily overwhelmed by the sheer complexity and size of their programs [25]. General-purpose programming languages make good use of features such as:

functions and subroutines; package modules; classes and objects; and more recently closures and iterators. Even so, these languages often have associated with them a set of operational baggage and idiomatic features that often obscure the essence of an application and can therefore make it unnecessarily complex to develop.

A relatively recent approach is to develop a domain-specific programming language (DSL) [5, 8, 10, 12] that offers a high-level set of mechanisms for the programmer to formulate ideas for a particular application domain of interest and body of knowledge. DSLs aim is to have the language focus concisely and specifically on just those aspects and concepts that are relevant to the particular problem domain. They are designed to hide away "boiler plate" code that is only an artifact of the underlying programming language and which does not contribute to the semantics of the application domain.

Many application domains including: simulations [14]; business systems; database systems; materials physics problems or other complex systems [16] could be supported in this way [8, 40]. In this present paper we develop a DSL for graph network analyses and numerical computations.

DSLs can be implemented [43] as either full-blown languages in their own right and with all the normal compiler [1] and builder environment apparatus to aid the programmer [24]. Although modern compiler generation tools [35, 37, 38] can aid the task, to develop such an "external DSL" requires considerable developmental effort and to construct a seamless programming environment a great deal of additional other computational apparatus such as normal arithmetic, logic, text and string handling features must also be implemented [31].

A lighter-weight approach is to use either pre-processor technology or other pre-existing features of a conventional programming language to support the addition of some macros or high-level language features. This means that the DSL features are effectively just mounted on top of the conventional language. This resulting language is known as an "internal DSL" and is often easier to develop but also can capitalise on existing libraries as well as the existing host language features and support apparatus. Internal DSLs are also sometimes known as embedded DSLs [20].

The DSL approach [33] is particularly powerful when you one is able to abstract some major set of operations and data structures together into a library framework and have them invoked by the programmer through memorable and compact programming language features. When a family of problems [2] can be identified and a few special cases are solved, then its often feasible and efficient to develop a DSL framework [23] that solves the whole class of problems rather than requiring each one to be solved separately and by hand. This obviously makes the DSL approach highly productive for tackling a range of related problems that share commonalities and patterns [36].

A fluent interface is one where syntactic features of the hosting language are used to good effect to construct an internal DSL that captures the jargon, the commands and other notions of the requisite application domain. We make use of modern Java [11] and language features such as foreach iteration [22] to implement a fluent interface for our graph and network computational framework. We are able to support graph algorithms [28] to compute component labelling [18]; path and loop identification and classification [17, 26]; and generation [27, 29] of a range of different synthetic graph data sets.

Building graph support software libraries is not new and there are several high quality systems available for C++ programmers [32, 39] that optimise for speed and for use of large graphs with distributed systems [19]. Systems using Java are still less common [34] and although systems such as Green-Marl [19] use an external DSL technique, to dat e no system ha s been reported using internal an DSL. This preset article concentrates on an internal DSL approach to a graph framework using a Java-based DSL. In particular we use Java iterators and associated language features to support a high level fluent interface to enable rapid expression of graph algorithms that traverse both nodes and arcs. The internal DSL approach avoids having to reimplement many normal language features that are required when developiung a full external DSL. We explore some examples of arithmetic and associated types required in the graph analysis examples given in Section 3.

Our article is organised as follows: In Section 2 we discuss the graph support framework and how we used various Java language capabilities [3, 42]. We discuss the fluent interface and the use of Java method -chaining in Section 3 and present some selected results using the system in Section 4. We give a comparison of necessary lines of code and discuss how the fluent interface system reduced these and the consequent code complexity in Section 5 and offer conclusions are areas for further work in Section 6.

2 Graph Computational Framework

'We show various Java code listing fragments to illustrate how we implemented the graph calculation framework (known as "Grapheon") and also how the fluent interface was supported through the use of iterators and other high level modern Java language features.

The code listed in Figure 1 shows the main data structure used for the graph computational framework. The individual fields in this and the other listings are commented as structural – when they hold the main graph structural definition or auxiliary when they are derived quantities that make it easier (faster or more elegantly coded) to implement the various computations. Two linked lists for

```
class Graph{
  // Structural:
  public List<Node> nodes = new Vector<>();
  public List<Arc> arcs = new Vector<>();

  // Computation Auxiliaries:
  boolean adjacency[][];
  int path[][];
  int next[][];
  double distances[][];
}
```

Figure 1: Data Structure for the Graph Framework.

the graph nodes and arcs are the primary data structures used. Other derived arrays are allocated memory only when needed and are used to augment some of the implementation methods.

```
class Node{
  // Structural:
  List<Arc> inputs  = new Vector<>();
  List<Arc> outputs = new Vector<>();

  // Convenience:
  List<Node> dsts = new Vector<>();
  List<Node> srcs = new Vector<>();

  // Decorative:
  int index     = 0;
  int mark      = 0;
  double weight = 1.0;
  String label  = "";

  // Computation Auxiliaries:
  int component   = 0;
  int betweenness = 0;
  int count       = 0;
  boolean visited = false;
  boolean blocked = false;
}
```

Figure 2: Data Structure for a Graph Node.

The code listed in Figure 2 shows the data structures used for each node in the graph. For development of experimental algorithms it is often easiest to have a choice of data representations but particularly using Java syntax it is neatest to have the node decoration properties – counts, visitation status and so forth – be part of the node data structure itself.

For iterating over nodes it is also convenient to support lists of each input and output Arc that connects the node in question, but also to have explicit lists of the neighbouring nodes. This means that for-each iterations can be expressed directly when this will neaten up expression of a computation algorithm such as for components, circuits and so forth as described in section 3. Internally each node has a definite integer index that will typically indicate the order it is generated or was inserted into the graph. We can therefore also track or loop over nodes in strict forward or reverse order.

```
class Arc{
  // Structural:
  Node src, dst;

  // Decorative:
  int index     = 0;
  int mark      = 0;
  double weight = 1.0;
  String label  = "";
}
```

Figure 3: Data Structure for a Graph Arc.

Similarly, we use a data structure for each Arc in the graph, as listed in Figure 3, that carries immediately accessible information about the nodes it connects as well as Arc properties such as weight and so forth. We therefore use some redundant ways of storing the graph information and use whichever is most suited to particular algorithms.

Being able to iterate over Nodes or Arcs or neighbours of a node is particularly useful to enable compact implementations of various algorithms. The code listed in Figure 3 shows how a Java Iterator can be implemented so that we can immediately apply a for-each iteration over a Depth-First Search (DFS) [6] starting at a particular node of the graph. A Breadth-First Search (BFS) is similarly implemented and these supplement iterations over all nodes; over all arcs or over all neighbours of a node, which are available as a direct consequence of our data structure lists.

Figure 5 shopws how a search is enabled by the iterators and the internal graph structures of the framework. The DFS or BFS Java iterator object is constructed on-the-fly in the for-each statement. Of necessity the constructor of the iterator must perform some setup to prepare the search/-traversal order. In principle, these iterator objects could be reused, by providing a reset method, so the internal traversal information is preservced and is reset ready for a re-traversal. We find that the most common use however is in this instantiate-and-discard pattern to support for-each traversals. Since the iterator object cannot be used for anything until it has calculated the correct traversal order, we have burdened the constructor with these calculations. This means that each iteration will take the same amount of computational time and that the first is not biased with the setup costs.

These data structures and traversal features can all be used to enable our Java-based fluent user interface which is described in Section 3 below.

```
public class DFS implements
          Iterable<Node>, Iterator<Node>{
  ...
  int iter = -1;
  int order[];

  public Node next()throws
                    NoSuchElementException{
    if( hasNext() )
      return graph.nodes.get(order[iter++]);
    else
      throw new NoSuchElementException();
  }

  public boolean hasNext(){
    return iter>=0 && iter<N;
  }

  public DFS(Graph graph, Node start){
    ...
    if( !start.visited )
      search(start);
    for( Node v : graph.nodes )
      if(!v.visited)search(v);
    iter = 0;
  }

  protected void search(Node start){
    LinkedList<Node> q = new LinkedList<>();
    q.addLast(start);
    while( !q.isEmpty() ){
      Node v = q.removeLast();
      if( !v.visited ){
        v.visited = true;
        order[count++] = v.index;
        for( int i=v.outputs.size()-1;
                 i>=0; i-- ){
          Node w = v.outputs.get(i).dst;
          if( !w.visited ) q.addLast(w);
        }
      }
    }
  }

}
```

Figure 4: Depth-first-search iterator using the Java Iterator interface requirements, in the DSL Graph Framework - Note the need for an explicit reverse iterator for DFS. Use as described in Figure 5.

3 Fluent Interface in Java

The code listed in Figure 6 shows an example of our fluent interface with a short complete program to read in a set of input graph files; report on the initial status of the composite graph; then carry out various computations on the graph before reporting and saving to a file. The Java

```
Node foundNode = null;
for( Node n : new DFS( graph, node0 ) )
  if( predicateFunction( n ) ){
    found = n ;
    break;
  }

Arc foundArc = null;
for( Node n : graph.nodes )
  for( Arc a : n.outputs ){
    if( predicateFunction( a ) ){
      foundArc = a ;
      break;
    }
  }
```

Figure 5: Use of the iterators for a search of the graph nodes or of its arcs.

method chaining technique is used so that each method in the fluent interface return "this" – a reference to the current graph object, upon which further computation methods can be invoked.

```
public static void main( String args[] ){

  Graph g = Graph.New( args )
  .setLogging(true)
  .report()
  .removeLeaves()
  .computeDegrees()
  .computeClusteringCoefficient()
  .computeAdjacency()
  .computeComponents()
  .computePaths()
  .computeBetweenness()
  .computeDistances()
  .computeCircuits()
  .report()
  .write( "composite.graph" )
  ;

}
```

Figure 6: Fluent interface to Java implementation of Graph computational framework. This would be run from a command line with "java Grapheon *.graph".

While the fluent approach above is intended as the primary way to access the graph framework, there are other possibilities enabled by having access to all the normal programming apparatus of a full capability programming language like Java.

Figure 7 shows code formulating a numerical experiment to investigate network robustness. This is a more compact way of carrying out the node culling experiments re-

```
for(int nCulled=0; nCulled<20; nCulled++){
  g.computePaths();
  g.computeBetweenness();
  g.removeNode( g.nodeWithMaxBetweenness );
  Output.printf("nCulled \t %d\n", nCulled);
  g.report();
}
```

Figure 7: Formulation of a numerical experiment to investigate network robustness.

ported by Hawick [15]. The most important node (with the highest betweenness centrality) is culled at each stage in a sequence and the network fragmentation and deterioration can be quantifiably measured.

```
Graph g = Graph.generateNP(100,0.01);

for( Node n : new BFS(g,0) )
  Output.printf("%d ",n.getIndex() );
Output.printf("\n");

for( Node n : new DFS(g,13) )
  Output.printf("%d ",n.getIndex() );
Output.printf("\n");

g.layoutAsClock();
g.render( "np.png" );
g.write( "np.graph" );
```

Figure 8: Verifying search-order collections iterators.

Networks need not necessarily be loaded from actual data sets in files. In some cases it is useful to be able to generate a set of synthetic graphs according to some pattern. The code listed in Figure 8 shows how our random graph generator is invoked as a factory method to produce an Erdos-Renyi random graph [9] of N nodes, each connected together with probability P. The code shows how we can traverse the generated graph in breadth-first or depth first order using the Iterators described above in Section 2. We also show how we have implemented auxiliary routines to render the graph in various ways as a portable network graphics (png) file for example.

Figure 9 shows how a computational experiment can be simply formulated using the DSL to count the number of connected components. In this case we use iteration over a collection of probability parameter values, as well as looping over a fixed number of samples. The Moments class is constructed with a simple interface to allow the separate collection of data and subsequent computation of average, standard deviation and any other desired statistical moments from the distribution of measured values.

In Figure 10 we show one of our more sophisticated computation methods in action. Counting the number of circuits or loops is not trivial [21] and reported work on this notes the complexity of the software and the necessary user interface to do so. We have reduced the code down to a single method call in our fluent interface to the framework.

```
int nSamples = 10;
int N = 1000;
double pVals[]={0.01,0.02,0.03,0.04,0.05};

for( Double p : pVals ){
  Moments cM = Moments.New("nComponents");
  for(int i=0;i<nSamples;i++){
    Graph np = Graph.New()
    .generateNP( N, p )
    .computeComponents()
    ;
    cM.add( np.nComponents );
  }
  cM.compute();
  Output.printf("%f \t %f \t %f\n",
                p, cM.average, cM.stdDev );
}
```

Figure 9: Experiment to measure number of network components in N,P graph as P varies..

```
Graph g = Graph.New()
                .generateNP( 50, 0.06 )
                ;

Circuits c = Circuits.New(g)
                     .setLogging(false)
                     .compute()
                     ;

Output.printf("nCircuits \t %d\n",
              c.nCircuits);

for( int i=0;i<=c.N;i++)
  Output.printf("%d \t %d\n",
                i, c.lengths[i]);
```

Figure 10: Experiment to histogram the loops (elementary circuits) in an N,P graph.

4 Selected Results

In this present article we focus on the framework and domain-specific language interface and not on the graphical interactive rendering that our system also supports. However, in Figure 11 we show one rendering of an N, P random graph with the nodes coloured according to their degree. Nodes are displayed equally spaced around a circle with Arcs shown as connecting lines. Various other colouring schemes according to other node properties are also

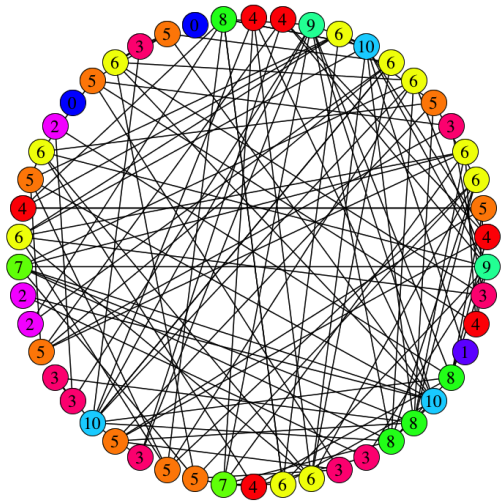

Figure 11: N,P generated random (directed) graph of 50 Nodes, P=0.06; with 127 arcs and mean degree of 5.08.

possible and individual components could be separately grouped or coloured.

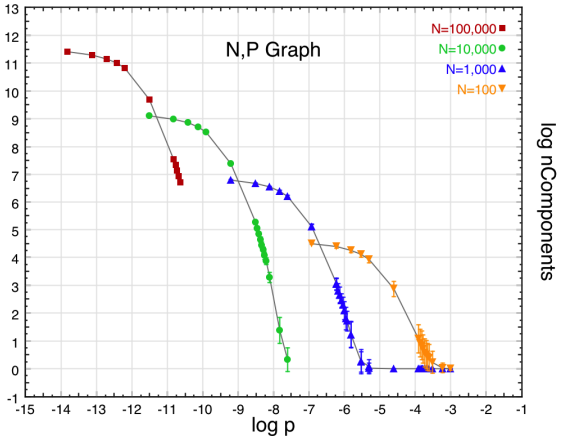

Figure 12: Numerical Results obtained from an experiment such as shown in Figure 9 to count components.

This generated graph is representative of the N, P graphs o f various sizes and connection probability values that we sampled to generate the plots in Figure 12. The number of connected components is shown measured over 10 samples at each size and probability and as can be seen there is a critical (sharp decline) in the number of components above which connection probability the graph tends to a single giant fully connected component in which all nodes are mutually reachable. Note that the steepness of the curves becomes sharper as the number of nodes in the graph is increased. We have plotted the curves on a log-log scale to show this effect and also the shift in critical P value - which is linked to the N/P ratio.

We can also exercise our circuits counting software and Figure 13 shows the histogram of the size distribution of

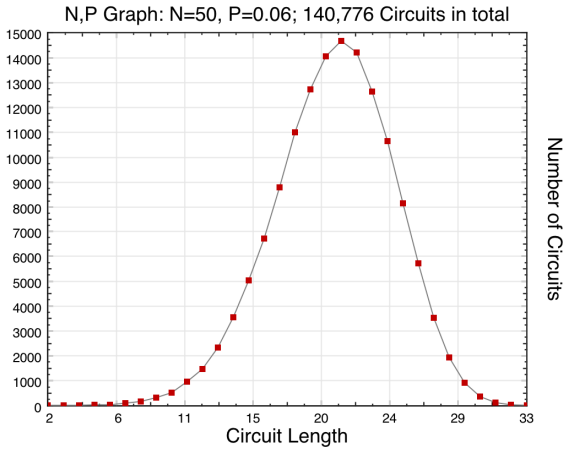

Figure 13: Numerical Results obtained from an experiment such as shown in Figure 10 to count circuits.

elementary circuits (closed paths) in an N, P graph of size $N = 50$. The number of circuits grows quite rapidly with N and so we only show a small sized system as an illustrative example. Note the characteristic circuit length given by the peak in the length distribution. Different sorts of generated graphs will have different distributions and characteristic signatures. One approach to understanding a particular network data set is therefore to measure these properties and try to match up the associated categorization signature with those known from generated graphs of the various parameter families.

5 Discussion

We have chosen to experiment with our own Java graph framework but the important ideas presented is the user interface provided by the DSL and fluent method chaining approach. In principle we can use any of the publicly available graph libraries to underpin such a fluent interface [19]. It then becomes a separable software engineering problem to implement the framework and the user interface. User calculation programs then are much more manageable in size and consequently have a lower complexity with all the usual advantages that accrue – easier testing more explicable code and so forth.

To make a reasonable assessment of the efficacy of both the fluent interfacing approach to graph calculations, we compare the lines of source code necessary to express various graph calculation problems with a D implementation and also with a C++ implementation. The D code was written circa 2009 and was able to make use of some early iterator ideas that were supported in that language [4]. The C++ implementation is from 2011 and was also able to make use of recent language features [41].

Table 1 indicates that our Java DSL 2012 shows approximately a factor of two reduction in lines-of-code (LOC).

Graph Metric Name	D LOC	C++ LOC	DSL LOC	Implementation Comment
Components	81	63	35	iterative
Paths	88	59	38	Floyd algorithm
Betweenness	-	48	26	requires paths
Newman	-	30	19	clustering coeff
Distances	93	64	27	all-pairs
Circuits	75	-	40	recursive
BFS	-	-	99	Iterator Object
DFS	-	-	37	over-rides BFS

Table 1: Lines of Java Code unique to each calculation, meaningful to ± 4 LOC. Java DSL is supported by $\approx 1,500$ lines of framework code; C++ and D versions required additional $\approx 3,750$ LOC in libraries.

More importantly however, the method-chaining approach gives a human-readable fluent interface that is reasonably close conceptually to the domain-specific jargon of graph calculations. It was also easier to separate out auxiliary calculations such as the statistical moments, that are not directly relevant to the graph algorithms, but which are very important to the compact and rapid expression of a numerical experiment.

The particular advantage of using 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 and we have made particular use of Java's Iterator facility to make for-each loops work over graph specific collections such as lists of Nodes.

Our interface can be used to invoke whatever framework underpins it and so computational performance is not a direct issue for the work in this article. Even with our own Java implementation of the experimental framework we were able to work with useful generated graph sizes of up to $N \approx 100,000$ in compute times of minutes on a modern desktop. This is not feasible for the more computationally demanding algorithms like circuit enumeration however.

We implemented our system so that although there was redundant duplication of structural graph information, the extra structures like adjacency arrays were only allocated memory when needed. Many of the algorithms can be elegantly implemented using just sparse lists. We observe that for this class of problem adding extra linear storage demands to nodes does not appreciably limit the problem size that is tackled. Making the Node class quite rich with additional lists to ease elegance of some of the algorithms was therefore a good design decision. Different frameworks will usefully adopt different strategies depending upon the problem domain and graph characteristics for which they are optimised.

There is scope for several other facilities and capabilities. We are presently working on compact ways to express graph mutation – method invocations to allow graphs to be filter and altered dynamically according to logical expressions concerning node, arc and other properties. We have focused on directed graphs and of course an undirected edge can be just represented as a pair of directed arcs. For some graph problems it is more natural to express the problem in terms of (undirected) edges and we also plan to implement some auxiliary routines to support that.

It may also 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 approach for incorporating automatic parallelism into the implementations [30].

The use of Java iterators has provided a mechanism to implement quite high level expressions to traverse both nodes and arcs of a graph using the for-each statement. It ought to be possible in principle to implement such a DSL interface on top of other Java graph libraries such as [34]. At the time of writing there are few large scale Java graoph libraries reported in the literature. There are several high quality C++ libraries such as Boost [39] and LEDA [32]. It would be possible in principle to use a Java "native method" approach and layer a Java interface on top of such libraries. We have chosen to concentrate on the iterator and fluent interface ideas within an internal DSL in this present paper.

Data-parallel or multi-threading parallelism ideas might be implemented to support a parallel code generator. This has the potential capability to generate almost all the boilerplate parallel code to support these calculations and hiding such infrastructure away from the ordinary complex networks domain user is attractive. An **external** DSL approach may be needed to take full advantage of this approach.

6 Conclusion

We have developed a fluent interface and Java-based **internal DSL** to a framework for graph network computations. We have developed a framework in Java that supports various forms of for-each iteration over nodes, arcs, neighbours, BFS and DFS traversals. Using this interface and framework we have developed implementations for computing connected components; circuits; pathways; distances and other static graph metrics. A range of complex network generation routines and file reading codecs have also been built using this interface.

We found that using multiple graph structural representations and adding redundant auxiliary extra storage for

nodes, such that the storage complexity was still linear in N did not appreciably limit the size of graph networks we could analyse. The extra used memory allowed more elegant and compact algorithmic formulations and was a worthwhile tradeoff for our system.

Our internal domain-specific language helps formulate numerical experiments to compute average and variational properties of random and scale free graphs based on statistical sampling over many randomly generated network realisations. The DSL also supports rapid formulation of experiments on particular real network data loaded from a file or originating from another application. Our fluent-interface could be implemented to use other graph libraries other than the experimental one we have constructed to test the interface ideas.

We have compared the lines-of-code needed to implement various experiments in our DSL+framework with other programmes built using other languages such as C++, D or Java without the framework. Our fluent system lowers the code complexity considerably by this metric.

There is scope for several further graph algorithms and computations to be implemented using the DSL and framework. This approach shows great promise as a software engineering technique for reducing code complexity and thus enhancing implementation feasibility and code verifiability. This is particularly useful for enabling sophisticated computational experiments in the field of complex networks investigation.

References

[1] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers Principles, Techniques and Tools. Addison-Wesley, second edn. (2007), iSBN 0-321-48681-1

[2] 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(2), 191–214 (April 2002)

[3] Bracha, G.: Generics in the java programming language. Tech. rep., Sun Microsystems (July 2004)

[4] Bright, W.: D Programming Language. Digital Mars (2008), www.digitalmars.com/d/

[5] Consel, C.: Domain specific languages: What, why, how. Electronic Notes in Theoretical Computer Science 65, 1 (2002)

[6] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction To Algorithms. MIT Press, 2 edn. (2001)

[7] van Deursen, A., Klint, P.: Little languages: little maintenance. Journal of Software Maintenance: Research and Practice 10(2), 75–92 (1998)

[8] van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Notices 35, 26–36 (June 2000), http://doi.acm.org/10.1145/352029.352035

[9] Erdös, P., Rényi, A.: On random graphs. Publicationes Mathematicae 6, 290–297 (1959)

[10] Fowler, M.: Domain-Specific Languages. No. ISBN 0-321-71294-3, Addison Wesley (2011)

[11] Freeman, S., Pryce, N.: Evolving an embedded domain-specific language in java. In: Proc. OOPSLA'06. pp. 855–865. Portland, Oregon, USA (22-26 October 2006)

[12] Ghosh, D.: Dsl for the uninitiated - domain-specific languages bridge the semantic gap in programming. Communications of the ACM 54(7), 44–50 (2011)

[13] Gould, R.: Graph Theory. The Benjamin/Cummings Publishing Company (1988)

[14] Hawick, K.A.: Engineering internal domain-specific language software for lattice-based simulations. In: Proc. Int. Conf. on Software Engineering and Applications. pp. 314–321. IASTED, Las Vegas, USA (12-14 November 2012)

[15] Hawick, K.A.: Water distribution network robustness and fragmentation using graph metrics. In: Proc. Int. Conf. on Water Resource Management (AfricaWRM 2012). pp. 304–310. No. 762-037, IASTED, Gabarone, Botswana (3-5 September 2012), cSTN-158

[16] Hawick, K.A., James, H.A.: Performance, scalability and object-orientation in discrete graph-based simulation models. In: Int. Conf. on Modeling, Simulation and Visualization Methods (MSV'05). pp. 25–31. CSREA, Las Vegas, USA (27-30 June 2005), iSBN 1-932415-70-X

[17] Hawick, K.A., James, H.A.: Enumerating circuits and loops in graphs with self-arcs and multiple-arcs. In: Proc. 2008 Int. Conf. on Foundations of Computer Science (FCS'08). pp. 14–20. CSREA, Las Vegas, USA (14-17 July 2008)

[18] Hawick, K.A., Leist, A., Playne, D.P.: Parallel Graph Component Labelling with GPUs and CUDA. Parallel Computing 36(12), 655–678 (December 2010), www.elsevier.com/locate/parco

[19] Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-marl: A dsl for easy and efficient graph analysis. In: Proc. 17th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012). London, UK. (3-7 March 2012)

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

[21] Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM Journal on Computing 4(1), 77–84 (March 1975)

[22] Kabanov, J., Hunger, M., Raudjarv, R.: On designing safe and flexible embedded dsls with java 5. Science of Computer Programming 76, 970–991 (2011)

[23] Karlsch, M.: model-driven framework for domain specific languages demonstrated on a test automation language. Master's thesis, Hasso-Platner-Institute of Software Systems Engineering, Potsdam, Germany (2007)

[24] 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(2), 247–264 (May 2010)

[25] Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., Turski, W.M.: Metrics and laws of software evolution - the

nineties view. In: Proc. 4th Int. Symp. on Software Metrics (METRICS'97). Albuquerque, NM, USA (5-7 November 1997), iSBN:0-8186-8093-8

[26] Leist, A., Hawick, K.A.: Circuits as a classifier for small-world network models. In: Proc. WORLDCOMP 2009 International Conference on Foundations of Computer Science (FSC 09) Las Vegas, USA. No. CSTN-003 (13-16 July 2009), http://www.massey.ac.nz/~kahawick/cstn/003/cstn-003.html

[27] Leist, A., Hawick, K.A.: Graph generation on gpus using dynamic memory allocation. In: Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11). pp. 229–235. No. PDP3939, CSREA, Las Vegas, USA (18-21 July 2011)

[28] Leist, A.: Experiences in Data-Parallel Simulation and Analysis of Complex Systems with Irregular Graph Structures. Ph.D. thesis, Massey University, Auckland, New Zealand (November 2011), http://hdl.handle.net/10179/2992

[29] Leist, A., Hawick, K.A.: A Small-World Network Model for Distributed Storage of Semantic Metadata. In: Kelly, W., Roe, P. (eds.) Proc. 7th Australasian Symposium on Grid Computing and e-Research (AUSGRID 2009). Conferences in Research and Practice in Information Technology (CRPIT), vol. 99, pp. 49–56. Wellington, New Zealand (21 January 2009), iSBN 978-1-920682-80-4

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

[31] Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.): Domain-Specific Program Generation. No. 3016 in LNCS, Springer (2003), ISBN 3-540-22119-0

[32] Mehlhorn, K., Naher, S.: The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press (1999)

[33] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (December 2005)

[34] Naveh, B.: Jgrapht java graph library. Web Site (2011), http://jgrapht.org/

[35] Parr, T.: The Definitive ANTLR Reference - Building Domain-Specific Languages. No. ISBN 978-0-9787392-5-6, Pragmatic Bookshelf (2007)

[36] Parr, T.: Language Implementation Patterns, Create Your Own Domain-Specific and General Programming Languages. No. ISBN 978-1-93435-645-6, Pragmatic (2010)

[37] Parr, T.: A functional language for generating structured text. Tech. rep., University of San Francisco (2006)

[38] Parr, T., Quong, R.W.: ANTLR: A Predicated LL(k) Parser Generator. Sogtware - Practice and Experience 25(7), 789–810 (July 1995)

[39] Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2001), iSBN-13: 978-0-201-72914-6

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

[41] Stroustrup, B.: The C++ Programming Language. No. ISBN 0-201-88954-4, Addison-Wesley, 3rd edition edn. (2004)

[42] Sun Microsystems: Java Programming Language. http://java.sun.com, last accessed September 2009

[43] Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis, University of Amsterdam (1997)