Using Hyper/J to implement Product-Lines: A Case Study

Roberto E. Lopez-Herrejon and Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712
{rlopez,batory}@cs.utexas.edu

Abstract. *Aspect-Oriented Programming (AOP)* is an emerging technology whose goal is to modularize concerns that may involve several classes. The purpose of this report is to describe how one of the main representatives of AOP, namely Hyper/J, was used to implement a simple yet illustrative product-line of graph algorithms.

1 The Graph Product Line (GPL)

The *Graph Product-Line (GPL)* [5] is a family of classical graph applications that was inspired by work on software extensibility [4, 8]. GPL is typical of product-lines in that applications are distinguished by the set of features that they implement, where no two applications implement the same set.[1] Also typical is that applications are modeled as sentences of a grammar. Figure 1a[2] shows this grammar, where tokens are names of features. Figure 1b shows a GUI that implements this grammar and allows GPL products to be specified declaratively as a series of radio-button and check-box selections.

The semantics of GPL features, and the domain itself, are straightforward. A graph is either `Directed` or `Undirected`. Edges can be `Weighted` with non-negative numbers or `Unweighted`. Every graph application requires at most one search algorithm: breadth-first search (`BFS`) or depth-first search (`DFS`); and one or more of the following algorithms [2]:

- **Vertex Numbering (**`Number`**):** Assigns a unique number to each vertex as a result of a graph traversal.
- **Connected Components (**`Connected`**):** Computes the *connected components* of an undirected graph, which are equivalence classes under the reachable-from relation. For every pair of vertices x and y in an equivalence class, there is a path from x to y.
- **Strongly Connected Components (**`StronglyConnected`**):** Computes the *strongly connected components* of a directed graph, which are equivalence classes under the reachable-from relation. A vertex y is reachable form vertex x if there is a path from x to y.

1. A *feature* is a functionality or implementation characteristic that is important to clients [3].
2. The grammar does not preclude the repetition of algorithms, whereas the GUI does.

(a)

```
GPL := Gtp Wgt Src Alg+;

Gtp := Directed | Undirected;

Wgt := Weighted | Unweighted;

Src := DFS | BFS | None;

Alg := Number | Connected | StronglyConnected
     | Cycle | MST Prim | MST Kruskal | Shortest;
```

(b)

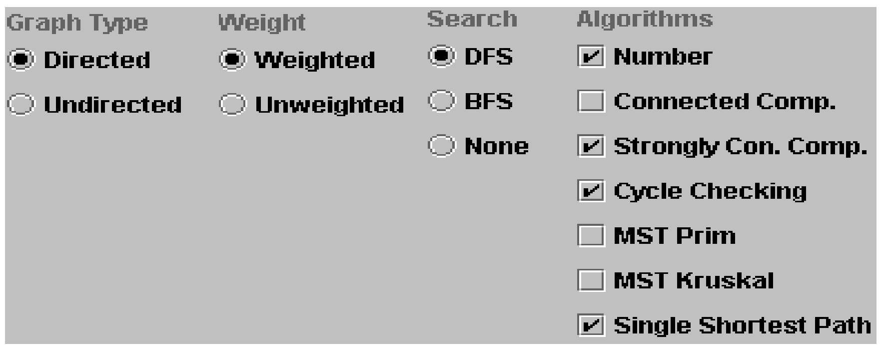

Figure 1. GPL Grammar and Specification GUI

- **Cycle Checking** (`Cycle`): Determines if there are cycles in a graph. A cycle in directed graphs must have at least 2 edges, while in undirected graphs it must have at least 3 edges.
- **Minimum Spanning Tree** (`MST Prim`, `MST Kruskal`)**:** Computes a *Minimum Spanning Tree (MST)*, which contains all the vertices in the graph such that the sum of the weights of the edges in the tree is minimal.
- **Single-Source Shortest Path** (`Shortest`)**:** Computes the shortest path from a source vertex to all other vertices.

A fundamental characteristic of product-lines is that not all features are compatible. That is, the selection of one feature may disable (or enable) the selection of others. GPL is no exception. The set of constraints that govern GPL features are summarized in Table 1.

A GPL application implements a valid combination of features. As examples, one GPL application implements vertex numbering and connected components using depth-first search on an undirected graph. Another implements minimum spanning trees on weighted, undirected graphs. Thus, from a client's viewpoint, to specify a particular graph application with the desired set of features is straightforward. And so too is the implementation of the GUI (Figure 1b) and constraints of Table 1.

Algorithm	**Required Graph Type**	**Required Weight**	**Required Search**
Vertex Numbering	Directed, Undirected	Weighted, Unweighted	BFS, DFS
Connected Components	Undirected	Weighted, Unweighted	BFS, DFS
Strongly Connected Components	Directed	Weighted, Unweighted	DFS
Cycle Checking	Directed, Undirected	Weighted, Unweighted	DFS
Minimum Spanning Tree	Undirected	Weighted	None
Single-Source Shortest Path	Directed	Weighted	None

Table 1. Feature Constraints

2 Graph Representation

While deciding how to represent our graphs, we recognized that there are a standard set of “conceptual” objects that are referenced by all graph algorithms: Graphs, Vertices, Edges, and Neighbors (i.e., adjacencies). Algorithms in graph textbooks define fundamental extensions of graphs, and these extensions modify Graph objects, Vertex objects, Edge objects, and Neighbor objects. Thus, the simplest way to express such extensions is to reify all of these “conceptual” objects as physical objects and give them their own distinct classes.

Therefore we represent a graph with these four classes:

- `Graph`: contains a list of `Vertex` objects, and a list of `Edge` objects.
- `Vertex`: contains a list of `Neighbor` objects.
- `Neighbor`: contains a reference to a neighbor `Vertex` object (the vertex in the other end of the edge), and a reference to the corresponding `Edge` object.
- `Edge`: extends the `Neighbor` class and contains the start `Vertex` of an `Edge`.

Edge annotations are performed by adding extra fields to the `Edge` class. This representation is illustrated in Figure 2. For example, Edge E1 connects vertex V1 to V2 with weight of 7.

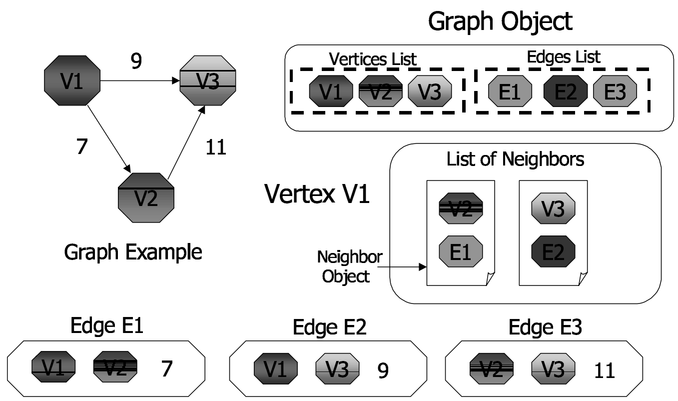

Figure 2. Edge and Neighbor List Representation Example

3 Hyper/J Implementation

We implemented GPL with the purpose of exploring how Hyper/J can be used to implement product-lines and to compare and contrast this implementation with the one that relies on mixin-layers with the ultimate goal of identifying the relationship between the two approaches.

Creating a Hyper/J application is a 3 step process [6, 7]:

- Define the hyperspace: consists of all the methods, classes, packages, etc. involved in the application.
- Define the concern mapping: break the hyperspace into pieces called hyperslices, and separate them as concern points along multiple dimensions.
- Define the hypermodule: specify how the hyperslices of the hyperspace are composed together.

Following this process, first each of the features from Figure 1a was implemented in a Java package that then was included in a hyperspace definition. For example, for the Number feature the package is named `GPL.Number` and it is included in the hyperspace with the following statement:

```
composable class GPL.Number.*;
```
[3]

Second, each of these packages was made to correspond to a hyperslice and each of them implements a different concern along the Feature dimension. Table 2 shows the names of the hyperslices and the concern they implement. For example, the following

3. This goes in the hyperspace definition file used for the composition.

mapping means that the package `GPL.Number` implements the Number concern in the Feature dimension.

```
package GPL.Number : Feature.Number
```
[4]

`Directed`	directed graph	**`Cycle`**	cycle checking
`Undirected`	undirected graph	**`MSTPrim`**	MST Prim algorithm
`Weighted`	weighted graph	**`MSTKruskal`**	MST Kruskal algorithm
`DFS`	depth-first search	**`Shortest`**	single source shortest path
`BFS`	breadth-first search	**`Transpose`**	graph transposition
`Number`	vertex numbering	**`Benchmark`**	benchmark program
`Connected`	connected components	**`Prog`**	main program
`StronglyConnected`	strongly connected components		

Table 2. Hyper/J hyperslices of GPL

Third, applications were defined for GPL (see Appendix).

Three hyperslices do not appear in Figure 1a: `Transpose`, `Benchmark`, and `Prog`. `Transpose` performs graph transposition and is used (only) by the `StronglyConnected` algorithm. It made sense to separate the `StronglyConnected` algorithm from `Transpose`, as they dealt with separate concerns. (This means that an implementation constraint in using the `StronglyConnected` aspect is that the `Transpose` aspect must also be included, and vice versa). `Benchmark` contains functions to read a graph from a file and elementary timing functions for profiling. `Prog` contains the main method. It creates the objects required to represent a graph whose elements are read from a file, and starts the execution of the algorithms.

A graph is implemented with the four classes mentioned above: Graph, Vertex, Neighbor, and Edge. Each hyperslice cross-cuts some of these classes, depending on the

4. This goes in the concern mapping file used for the composition.

functionality they implement. For example all the algorithmic hyperslices cross-cut the Graph class by adding the *run* method that executes the algorithm implemented by that particular hyperslice. This is illustrated in Figure 3 for the Number hyperslice.

```
package GPL.Number;

public class Graph {

        // Executes Number Vertices
        public void run(Vertex s) {
                NumberVertices();
        }

        public void NumberVertices() {
                GraphSearch( new NumberWorkSpace());
        }

        // STUB comes from a search feature
        public void GraphSearch( WorkSpace w) { }
}
```

Figure 3. Run method in Number hyperslice

By definition, a hyperslice has to be declaratively complete [6], that is, it must declare everything to which it refers. This requirement creates in GPL a stub proliferation problem. Recall, from Table 1, that the Number feature requires a search method, named GraphSearch which is implemented by either of two search hyperslices, BFS and DFS[5]. Notice in Figure 3 that in order to satisfy the declarative completeness property, a stub for GraphSearch was added. This problem is aggravated in GPL by the fact that a hyperslice usually refers to units from several other hyperslices, and also because some of the stubs that need to be added are variable or data members. The use of interfaces or abstract classes in GPL cannot alleviate this problem for the following reasons:

- Identifying and defining an interface takes at least the same effort as to write the stubs.
- Some stubs are variables or data members and therefore cannot be defined in interfaces.
- Some hyperslices create new objects. If these objects are instances of a class used for declarative completeness obviously such class cannot be abstract.

An application member of this product-line is defined by a hypermodule[6,7]. Special care should be taken in the order in which the hyperslices are composed. As an example, consider a family member whose hypermodule definition contains the Strongly-

5. Both search algorithms work on WorkSpace objects. NumberWorkSpace extends this class and customizes the search for the vertex numbering algorithm.

Connected hyperslice that requires the Transpose hyperslice (see Table 1), which implements the `ComputeTranspose` method. To satisfy the declarative completeness property, the StronglyConnected hyperslice declares a stub for this method that returns a null pointer, as illustrated in Figure 4. For the composition to run correctly Transpose hyperslice has to appear after StronglyConnected in the hypermodule definition so that the value assigned in the `StrongComponents` method is that computed in the Transpose hyperslice. If the Transpose hyperslice appears before the StronglyConnected one, the value assigned in `StrongComponents` method will be a null pointer (coming from StronglyConnected itself), that later on the execution of the program will cause a null pointer exception.

```
package GPL.StronglyConnected;

public class Graph {

        // STUB comes from Transpose hyperslice
        public Graph ComputeTranspose(Graph the_graph) {
                return null;
        }

        // Executes Strongly Connected Components
        public void run(Vertex s) {
                Graph gaux = StrongComponents();
                ...
        }

        public Graph StrongComponents() {
                ...
                // Compute the transpose of G
                Graph gaux = ComputeTranspose((Graph)this);
                ...
        }
}
```

Figure 4. StronglyConnected hyperslice

The hyperslices were composed with the default relationship *mergeByName*, with the exception of the method addAnEdge of the Weighted hyperspace which creates Edge objects with weights that overrides the behavior of that method in the Directed or Undirected hyperslices where weightless Edges are created. The reason for this exception is that: Prog hyperslice calls the method addAnEdge for each Edge object that needs to be added to the graph, if no overrides relationship is used each Edge will be effectively added to the graph twice, once in the Weighted hyperslice and once in the Directed or Undirected hyperslices. See ExampleB and ExampleC in the Appendix for a detailed example.

4 Findings

In product-line designs it is the case that not all syntactically valid composition of features are semantically valid. For example, consider the case where the Hyper/J programmer overlooked the constraint that the Cycle hyperslice requires the DFS hyperslice (see Table 1), and instead wrote down BFS in the composition files. Hyper/J would compose the hypermodule without any trouble, since both search hyperslices define and use the same data members and methods. However, evidently, the outcome of the execution of the algorithm will be incorrect.

The legal compositions of features in Table 1 are defined by simple constraints called *design rules* [1]. In Hyper/J there is no support for design rules, that is, the programmer has to manually select all the files necessary to create a new member and include them in the correct order in the hypermodule, this activity is complex and error prone even for small product-lines like GPL.

Declarative completeness of the hyperslices introduces the stub problem that is pervasive in GPL, most of the hyperslices present the problem, and sometimes it is not easy to deal with. Identifying the units that are missing and what hyperslices they come from is necessary for introducing their corresponding stubs manually (copy and paste) in the hyperslices that require them, which is a laborious task that definitely calls for an adequate tool support.

5 Appendix

The jar file associated to this report contains the entire source code of GPL and three examples located in the Examples directory. Those are:

- **ExampleA:** Prog, Benchmark, Cycle, Number, DFS, Undirected.
- **ExampleB:** Prog, Benchmark, MSTKruskal, MSTPrim, Cycle, Number, Connected, DFS, Weighted, Undirected.
- **ExampleC:** Prog, Benchmark, Shortest, Transpose, Cycle, Number, StronglyConnected, DFS, Weighted, Directed.

Each example has its own concern mapping, hyperspace definition, and hypermodule. For example for ExampleA those files are named *GLPA.cm, GPLA.hs*, and *GPLA.hm* respectively. The hypermodule and hyperspace definition of ExampleA are shown in n Table 3.

To compose the programs use the normal Hyper/J way. To run an application type:

```
java GPL.Prog.Main ..\BENCH\MSTExample.bench v0
```

The first argument is the bench file that you want to use as input for your application, so you have to provide the corresponding path to reach it. The second argument is the starting vertex that some algorithms require to begin the execution from.

Hypermodule	Hyperspace definition

`hypermodule GPLExampleA`
` hyperslices:`
`   Feature.Prog,`
`   Feature.Benchmark,`
`   Feature.Number,`
`   Feature.Cycle,`
`   Feature.DFS,`
`   Feature.Undirected;`
 	
` relationships:`
`   mergeByName;`
	`hyperspace GPLHyperspace`
	` composable class GPL.Undirected.*;`
	` composable class GPL.DFS.*;`
	` composable class GPL.Number.*;`
	` composable class GPL.Cycle.*;`
	` composable class GPL.Benchmark.*;`
	` composable class GPL.Prog.*;`

Table 3. Hypermodule and Hyperspace Example

As output the application displays the final values of the fields of all the Vertex and Edge objects in the graph, along with the time it took to execute the entire application.

6 References

[1] D. Batory and B. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, February 1997.

[2] T.H. Cormen, C.E. Leiserson, and R.L.Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[3] M. Griss, “Implementing Product-Line Features by Composing Component Aspects”, *First International Software Product-Line Conference,* Denver, Colorado., August 2000.

[4] I. Holland. “Specifying Reusable Components Using Contracts”, *ECOOP 1992*.

[5] R. E. Lopez-Herrejon, D. Batory. “A Standard Problem for Evaluating Product-Line Methodologies”. Third International Conference on Generative and Component-Based Software Engineering (GCSE), September 2001, Erfurt, Germany.

[6] Harold Ossher and Peri Tarr, “Multi-dimensional separation of concerns and the Hyperspace approach.” In Software Architectures and Component Technology (M. Aksit, ed.), 293-323, Kluwer, 2002.

[7] P Tarr, H. Ossher. “Hyper/J User and Installation Manual”. IBM Corporation, 2001. http://www.research.ibm.com/hyperspace.

[8] M. VanHilst and D. Notkin, “Using C++ Templates to Implement Role-Based Designs”, *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.