

An Object-oriented Design Methodology

Greg Sullivan, Karl Lieberherr

September 27, 1993

Abstract

This paper presents a methodology for software analysis and design and an associated graph-based representation. The methodology addresses the specification of both static and dynamic aspects of an application. The representation includes object-oriented data modeling features. The specifications are modular, reusable, and adaptive – that is repercussions from underlying model modifications are limited. There is a straightforward process for generating code to implement the specifications in an object-oriented target language.

Keywords: Software engineering, methodology, object-oriented analysis, design and programming, reusable software, class dictionaries, graph-based programming, adaptive programming.

This work is supported in part by the National Science Foundation under grants CCR-9102578 (Software Engineering) and CDA-9015692 (Research Instrumentation).

1 Introduction

This paper presents a methodology for developing object-oriented software. The methodology includes a relatively simple representation, encompasses both the data and procedural aspects of the application, and can be supported by code generation facilities. Most of the constructs in the methodology are presented with examples of corresponding automatically generated implementations in C++. An important aspect of this paper and the methodology is that the generation of code from the abstract representations of the methodology is fairly straightforward. Significant work on graph-based data representation and programming, as well as implementing such specifications in C++, has been done as part of the Demeter research project at Northeastern University.

The relationships between classes in an application are represented in our methodology by edges between vertices (vertices correspond to classes) in a graph called the *application schema*. The edges may represent inheritance relationships, part-of relationships, or derived (functional) relationships between classes. The application schema is then available to be used by *propagation patterns*, which provide a powerful, robust method of specifying behavior as traversals of the runtime application schema. Propagation patterns support *adaptive programming*, which is the concise specification of application functionality while allowing for evolution of the underlying schema.

1.1 Extended Example - Train Scheduling

We will present the elements of our methodology in the context of a train scheduling application. Assume that there is a network of bidirectional train tracks serving some area of the country. We are to implement a system which schedules trains on routes through the network.

To schedule a train, a train operator submits a request to schedule a train trip from an origination point to a destination point, including time of departure from the origination. The scheduling system returns a departure time, route and time of arrival. A route consists of a series of track segments, with possibly a wait between segments.

2 Representation of Schemas

The primary data structure in our methodology is a graph called the *application schema*. A schema consists of vertices, which represent application classes, and edges between the vertices, which represent relations between the classes. There are three types of vertices: construction, alternation, and repetition. There are four types of edges: relation, function, alternation, and repetition. More concisely, then, a schema definition S is a seven-tuple,

$$S = \langle V_c, V_a, V_{rep}, E_{rel}, E_f, E_a, E_{rep} \rangle .$$

V_c , V_a , and V_{rep} are the sets of construction, alternation, and repetition vertices, respectively. E_{rel} , E_f , E_a , and E_{rep} are the sets of relation, function, alternation, and repetition edges, respectively.

All vertices have a single attribute, *label*. Which vertex set (V_c , V_a , or V_{rep}) a vertex is in determines constraints on which types of edges may be incident on the vertex and what sort of interfaces are generated.

All edges are four-tuples,

$$E = \langle label, signature, over, code \rangle .$$

As for vertices, which edge set an edge is a member of (E_{rel} , E_f , E_a , or E_{rep}) determines constraints and generated interfaces.

The signature element of an edge four-tuple is of one of the following four forms:

1. $from \times to$
2. $from \rightarrow to$
3. $from \Rightarrow to$
4. $from \rightsquigarrow to$

Where *from* is a vertex name or a cross product of vertex names and *to* is a vertex name. In the first case, the edge is a relation, a subset of the indicated cross product, and is in E_{rel} . The second case indicates that the edge is a function and is in E_f . The third case indicates an alternation edge from *from* to *to*, which states that the *to* class is a subclass of the *from* class and that the edge is in E_a . The final case indicates that the edge is a repetition edge, that the *from* class is a collection of instances of the *to* class, and that the edge is in E_{rep} . In the first two cases (relation and function edges), *from* may be a cross product of vertex names, whereas in the last two cases (alternation and repetition), *from* may only be a single vertex name. For alternation and repetition edges, the *label* element of the edge tuple may be empty.

The vertex names in a signature may be annotated with names so that code fragments may reference them. For example:

$$(net : Network \times request : TripRequest) \rightarrow the_trip : Trip$$

is a signature for a function edge from *Network* and *TripRequest* to *Trip*. Any code attached to the edge definition used to implement this function may refer to *net* and *request* as input parameters and may set *the_trip* as an output parameter. If the return class has a name bound to it, a new instance of the return class is created at the start of the edge traversal, and the return name (*the_trip* in the preceding example) is bound to it.

The *over* and *code* elements of an edge specify the procedural logic to be invoked when the edge is traversed. Specifying edge behavior will be discussed later.

2.1 Construction Vertices

The most common type of vertex in a schema is called a *construction vertex*, and it is represented by a rectangle with a label. Construction vertices represent instantiable classes in the application. In our train scheduling example, obvious construction vertices from the application domain include **Train**, **Track**, **Trip**, and **Location**. Figure 1 includes construction vertices for classes *Network*, *Track*, and *Loc*.

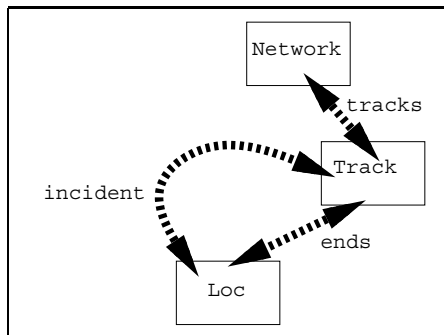



Figure 1: Initial Schema, Using Relation Edges, for Track Network

2.2 Relation Edges

The most general type of edge used to connect two or more vertices is the *relation edge*, drawn as . The relation edge labelled *incident* in Figure 1 between vertices *Loc* and *Track* establishes a binary relation of $\langle Loc, Track \rangle$ pairs. To decide which tracks are incident on a given location, we can search the **incident** relation for any pairs with the given location as their *Loc* element. Likewise, we can also determine which locations correspond to the endpoints of a track by searching the relation for the pairs with the *Track* element matching the given track. A simple first cut at a schema for a network of tracks is given in Figure 1.

The relation edges defined in Figure 1 have the following labels and signatures (separated by colons):

$$\begin{aligned} tracks &: Network \times Track \\ ends &: Track \times Loc \\ incident &: Track \times Loc \end{aligned}$$

While Figure 1 has only binary relations, the method allows n-ary relations. For example, we could define the ternary relation edge *allowedTrains* : $Train \times Track \times Time$, as shown in Figure 2. Note the *join node*, drawn as \bullet , which connects all n relation edges in an n-ary relation.

Consider a relation edge r with signature $from \times to$ as corresponding to the mathematical notion of a relation,

$$r \subseteq from \times to.$$

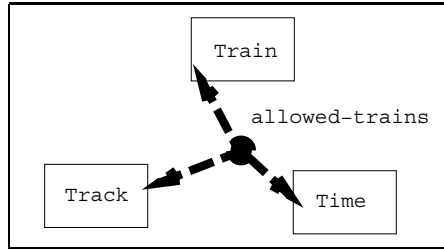


Figure 2: A ternary relation

2.3 Function Edges

Many of the relations between construction classes are in fact functions, where a single instance of a source class maps to at most one instance of a target class. We represent a function between classes as a directed, labelled *function edge* between vertices in the schema, drawn as $\xrightarrow{\text{label}}$. A function edge models what is often referred to as an “a-part-of” relationship between classes, although it is important to keep in mind that the edge could be implemented either as a function or as an instance variable. The domain of the function is the source class of the edge, and the range of the function is the target class of the edge. Figure 3 is a refined version of the schema in Figure 1 and includes function edges labelled *tracks*, *ends*, and *incident*, which define the following functions:

$$\begin{aligned} \textit{tracks} &: \textit{Network} \rightarrow \textit{List} < \textit{Track} > \\ \textit{ends} &: \textit{Track} \rightarrow \textit{OrdPair} \\ \textit{incident} &: \textit{Loc} \rightarrow \textit{List} < \textit{Track} > \end{aligned}$$

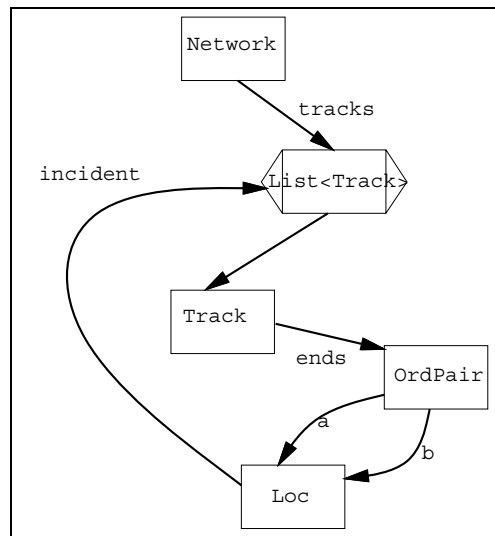


Figure 3: Refined Schema for Track Network

In general, a function is not limited to a single domain class. A function consists of a set of domain classes (parameters) and a target, or range, class. For example, if a track and a location

determine another location (i.e. a track and the location of one end determine the location of the other end), we have a function

$$\text{otherEnd} : \text{Track} \times \text{Loc} \rightarrow \text{Loc}$$

which is represented graphically in Figure 4. Note that the join node (\bullet) for function edges collects all incoming edges and emits a single outgoing edge to the range vertex.

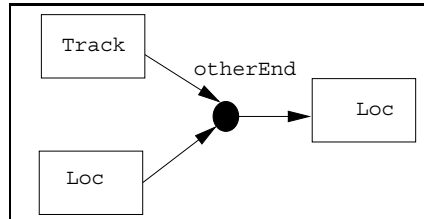


Figure 4: A function with multiple domains

A function edge is considered a *specialization* of a relation edge. Both correspond to n-ary relations. The fact that a relation edge has been specialized to a function edge imposes constraints on membership in the relation – namely that there exists exactly zero or one instance of the target class for any given tuple of instances of the domain classes. Having declared an edge to be a directed function edge also refines the interfaces that will be supported by the generated class definition code. For example, in Figure 3, there will be an interface named `tracks` which returns a `List<Track>` object given a `Network` object, but there will be no support for returning a `Network` object given a `List<Track>` object.

2.4 Repetition Vertices and Edges

It may be that a relation between classes is one-to-many, rather than the more general many-to-many relationship embodied by relations. We represent a one-to-many relation between classes by using a combination of a function edge, a *repetition vertex*, and a *repetition edge* in the schema. A repetition vertex, drawn as $\langle \square \rangle$, represents what is often also called a *container class* or *collection class*. The representations of several other methodologies use special edge types or edge labels to indicate one-to-many relationships between classes. We chose to use edges to special vertex types because this corresponds with a typical implementation strategy – i.e. using collection classes to implement one-to-many relationships.

Figure 3 contains a repetition class labelled `List<Track>` and a repetition edge from `List<Track>` to `Track`. This repetition vertex, its single repetition edge, and the target vertex of the repetition edge represent a collection of `Track` objects.

The combination of a function edge to a repetition class and an edge from the repetition class to a target class is another specialization of a generic relation edge. Using a repetition class and edge does not constrain membership in the named relation, but it does affect the supported interfaces to the relation. For example, from Figure 3, there will be interfaces (in the generated

code) which return a list of all tracks in a given network, but there will be no support for finding all networks of which a given track is a member.

There are several reasons to use function edges and repetition vertices rather than simple relation edges when specifying a schema. The use of these finer-grained constructs allows for more efficient implementations of the class structures to be generated automatically. Also, a more intuitive interface to classes may be presented than one based strictly on relations. If the relationship between two classes is specified using a relation edge, all interfaces must be in terms of sets of tuples, which is quite general but unwieldy.

It is worth noting that the behavior expressed by function edges, repetition vertices, and repetition edges is contained within that presented by simple relation edges. At the implementation level, any behavior implemented on top of repetition classes and edges may also be implemented on top of simple relation edges and construction vertices. The interface to the schema may be more natural and convenient if the schema takes advantage of repetition vertices and edges, though.

2.5 Alternation Vertices and Edges

To represent inheritance relationships in a schema, we use *alternation vertices*, drawn as \diamond , and *alternation edges*, drawn as \blacktriangleright . As an example of an alternation vertex, suppose that some of the tracks in the network are one-way. To represent this, the **Track** class becomes an alternation vertex with two child classes, **1Way** and **2Way**. See Figure 5 for a representation of the **Track** alternation vertex. An alternation edge from an alternation vertex to another vertex can be read as *is a superclass of*.

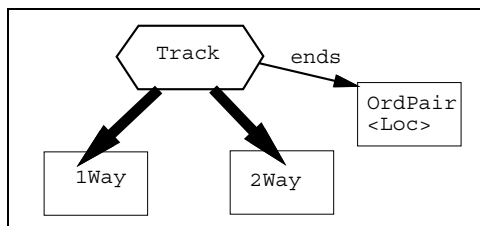


Figure 5: Track as an Alternation Class

Alternation edges must be sourced by alternation vertices. At most one alternation edge can connect an alternation vertex to another vertex (either a construction vertex or another alternation vertex), and therefore alternation edges are not labelled. Formally, an alternation edge from an alternation vertex to a construction (or alternation) vertex may be considered as establishing a relation. For example, the alternation edge from *Track* to *1Way* in Figure 5 establishes the relation

$$TrackHasAlternative1Way \subseteq Track \times 1Way$$

The relation is, in fact, a function:

$$1WayIsaTrack : 1Way \rightarrow Track$$

An instance of *1Way* may always find its parent instance of *Track* by invoking the *1WayIsaTrack* function on itself. In this way, alternation edges are another sort of refinement of relation edges. Constraints on entry into the relation are that a given *1Way* object may be represented at most once. In fact, all *1Way* objects must be represented in the relation.

Inheritance may be implemented in terms of relations. A method invoked on an instance of a child class but implemented in a parent class may be passed to the correct parent instance by first invoking the appropriate **Isa** function and then invoking the method on the returned parent instance.

A more formal presentation of alternation edges can be found in [LX93].

3 Adaptive Programming Using Edges

Once relationships between classes are codified as edge definitions, those edges become available to *propagation patterns*. Propagation patterns are an important element of *adaptive programming*, which allows behavioral specifications to adapt to changes in the underlying class structure.

Adaptive software and propagation patterns are described extensively in [Lie94], and we will give only a brief overview of them in this paper. One of the important insights of adaptive programming is that many of the algorithms required to implement application behavior involve traversing an application's run-time object graph. For example, if we want to calculate the total length of tracks in our train track network, we need to start at an instance of the *Network* class and find all instances of the *Track* class. The details of how one finds all of the actual *Track* objects in the network is of secondary importance and can be automated.

A propagation pattern that we might use to calculate the total length of all tracks in a network is:

```
*from* Network  
*to* Track
```

This propagation pattern will correctly traverse the runtime object graph if the application schema is as in Figure 6 (A). More significant, though, is that the above propagation pattern remains correct if the application schema evolves to that depicted in Figure 6 (B).

Adaptive programming and edges interact in two important ways:

1. Propagation patterns rely on paths in the application schema to get from the *from* classes to the *to* classes of a propagation pattern. These paths may contain edges which represent derived relationships between classes.
2. Edges which represent derived relationships between classes (as opposed to relationships implemented with instance variables) may use propagation patterns to specify the implementation of the edge behaviors.

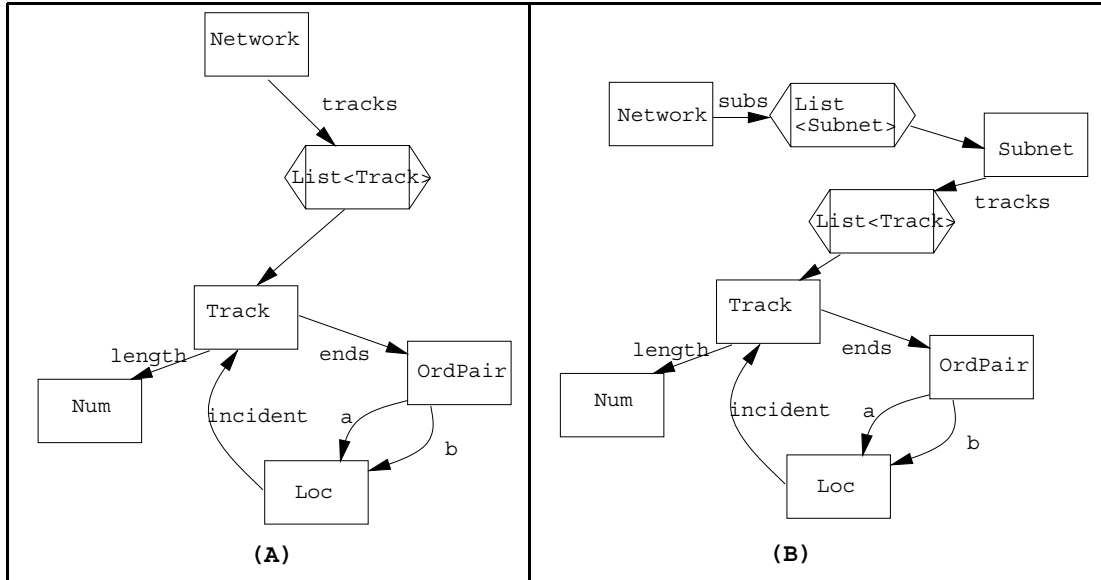


Figure 6: Network Schemas, Initial (A) and Enhanced (B)

Thus there is a significant synergy between the concepts of adaptive programming and derived edges, with each influencing and supporting the other.

4 Interfaces Induced by Schemas

As mentioned in the introduction, an important aspect of our methodology is that it is possible to generate useful application code from the graphical representations used. Vertices in schemas represent classes in the application, and edges induce methods of those classes. The different types of edges result in different sets of interfaces being generated. Following are the most important of the interfaces generated by different combinations of vertex and edge types. All application code will be presented in C++, with apologies to all who would prefer some other language.

4.1 Interfaces from Relation Edges

A relation edge labelled r from vertex $V1$ to vertex $V2$ results in the following public interface declarations:

<i>interface</i>	<i>comments</i>
Relation* V1::r();	return (a pointer to) the relation r
Relation* V2::r();	return (a pointer to) the relation r

4.1.1 Interfaces to Relation Objects

A *Relation* object, such as is returned by the `V1::r()` and `V2::r()` methods above, has the following interfaces:

<i>interface</i>	<i>comments</i>
Status Relation::add_tuple(int arity, ...);	adds a tuple to relation. ... is a list of length <i>arity</i> containing values to be inserted as elements of the new tuple
void* Relation::match(int arity, int n, ...);	the ... is a list of length <i>arity</i> which specifies a tuple to match. NULL values match anything. The n^{th} element of the 1 st matching tuple is returned. NULL is returned if no match.
Bag* Relation::matchall(int arity, int n ...);	the ... is a list of length <i>arity</i> which specifies tuples to match. NULL values match anything. An unordered collection (a bag) of the n^{th} elements of all matching tuples is returned. NULL is returned if no match.

For example, given a relation edge labelled *ends* between vertices *Track* and *Loc*, which establishes the relation:

$$\text{ends} \subseteq \text{Track} \times \text{Loc},$$

one could write the following C++ code fragment to return a bag of all *Track* objects incident on a given location `loc`:

```
return loc->ends()->matchall(2, 1, NULL, loc);
```

Note that when writing code, the ordering of elements in tuples is important and is determined by the order in the textual definition of the relation. For example, if the textual representation of the *allowedTrains* relation is

$$\text{allowedTrains} \subseteq \text{Train} \times \text{Track} \times \text{Time}$$

then the elements in the triples of the relation will be ordered as

$$\langle \text{Train}, \text{Track}, \text{Time} \rangle .$$

4.2 Interfaces from Function Edges

A function edge labelled *f* from source vertex *S* to target vertex *T* results in the following public interface declaration in C++:

<i>interface</i>	<i>comments</i>
T* S::f();	method f of class S returns a pointer to an instance of class T

A function edge labelled f from source classes $S1$ and $S2$ to target class T results in the following public interfaces:

<i>interface</i>	<i>comments</i>
$T^* S1::f(S2^*);$	method f of $S1$ returns a T^* given an $S2^*$
$T^* S2::f(S1^*);$	method f of $S2$ returns a T^* given an $S1^*$

4.3 Interfaces from Repetition Vertices and Edges

A repetition class Rep with a repetition edge to class Elt will result in the following interfaces:

<i>interface</i>	<i>comments</i>
Status $Rep::add(Elt^* e);$	add a new Elt to collection
Iterator< Elt > $Rep::iterate();$	return an Iterator, for iterating over collection
Bool $Rep::find(Elt^* e);$	returns TRUE if this Rep contains e (uses $==$)
$Elt^* Rep::nth(int n);$	returns n th element of this Rep ; NULL if length $> n$
int $Rep::length();$	returns number of elements in Rep

An $Iterator<Elt>$ object supports the following methods:

<i>interface</i>	<i>comments</i>
$Elt^* Iterator<Elt>::current();$	return current element of collection
Status $Iterator<Elt>::next();$	point to next element of collection. Return EOC if no next element
Status $Iterator<Elt>::goto(int n);$	point to n th element of collection

4.4 Declarations due to Alternation Vertices and Edges

If there is an alternation edge from parent vertex P to child vertex C , there are no new public interfaces created, but the definition of the child class is affected in C++. The class definition will now start with a line such as:

```
class C public P {
    ...
}
```

5 Implementation of Relations – Sets, Instance Variables, or Methods

Given that there is a function

$$ends : Track \rightarrow OrdPair,$$

we want interfaces available to the application programmer as described in the previous section. A typical example is the `ends` method, as used in the following code:

```
an_ordPair = a_track->ends();
```

The `ends` method of the `Track` class, and other relation and class-oriented methods, could all be implemented on top of any of the following three data structures:

Set The *ends* relation could be maintained as a set of ordered pairs of *Track* and *OrdPair* objects. The `ends` method would then be implemented by traversing the set looking for a pair with its *Track* element equal to the value of `a_track`.

Instance Variable The *ends* relation could be maintained as an instance variable of *Track*, with type `OrdPair*`. The `ends` method would then be implemented by simply returning the instance variable.

Method It is certainly possible that the instance of *OrdPair* returned by the *ends* relation given a *Track* object can be computed purely procedurally, by examining other available information. In this case, maintaining an instance variable or set would be redundant (but possibly better performing). For a relation to be implemented procedurally, a chunk of code must be supplied which, given an instance of *Track*, will return an instance of *OrdPair*.

The decision as to which implementation is actually used can be deferred as long as desired. A prototyping system may implement all edges as sets, which would be inefficient but simple to implement and change. Later in the design cycle, indications as to which implementation strategy is preferred can be given on an edge by edge basis.

6 Edge Definitions – Behavioral Specification

Any processing in an application corresponds to the traversal of edges in the application's schema. For example, if scheduling a train consists of submitting a trip request and receiving a filled-out *Trip* object, that process would correspond to an edge labelled *schedule* from the *TripRequest* vertex to the *Trip* vertex. The source vertex or vertices of an edge represent information necessary to perform the computation, and the target vertex of the edge represents the return value (an instance of the vertex's class) of the computation.

Recall that a schema edge is a four-tuple,

$$E = \langle label, signature, over, code \rangle .$$

The *signature* of the edge determines whether an edge is a relation, function, alternation, or repetition edge, namely whether the edge is in E_{rel} , E_f , E_a , or E_{rep} .

When the edge named *label* is invoked at runtime, it will return a (possibly newly allocated) instance of the target class of the edge. When the edge is invoked, the *over* traversal subschema associated with the edge is traversed. During the course of the traversal of the *over*

subschema, various code fragments, which are elements of the *code* edge element are invoked at the appropriate execution points of the traversal.

Note that alternation edges, which have no labels, are invoked when the target class of the alternation edge is instantiated. Repetition edges, which also have no labels, are invoked when a repetition class is iterated over.

6.1 The Traversal Subschema

Figure 7 is a definition for the *schedule* edge, which is given a *TripRequest* object and a *Network* object (the network of tracks) and returns a new instance of a *Trip* object.

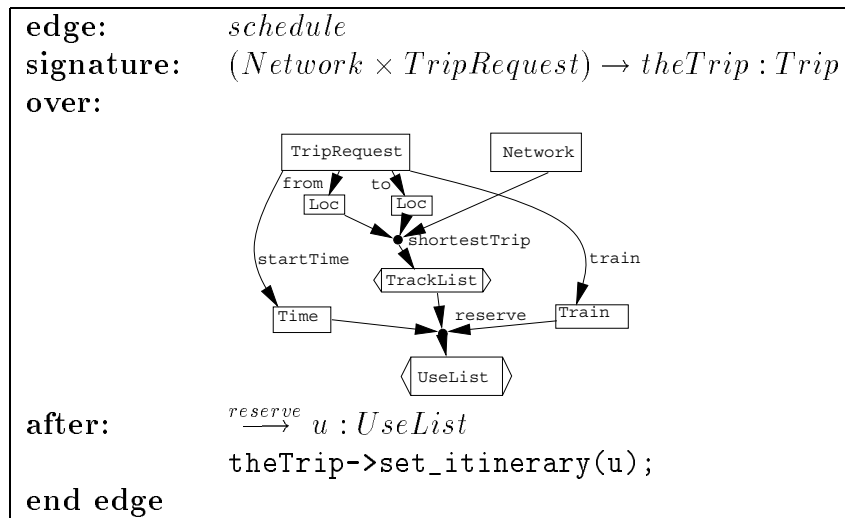


Figure 7: Definition of the schedule function edge

The *over* clause in Figure 7 is a graphical representation of the *traversal schema* which is to be traversed in order to produce the desired *Trip* object.

A *traversal schema* is a seven-tuple,

$$S_t = \langle V_c, V_a, V_{rep}, E_{rel}, E_f, E_a, E_{rep} \rangle .$$

which is the same as for application schemas. The primary difference between application schemas and traversal schemas is that vertices in application schemas have a single attribute, *label*, whereas in traversal schemas, vertices are a pair of attributes, $V = \langle label, index \rangle$. In traversal schemas, the same vertex label may appear more than once, with each occurrence having a different associated index attribute. In fact, all vertices in a traversal schema except join vertices (used for n-ary function or relation edges) may have **at most one incoming function, relation, or repetition edge**.

[what about multiple incoming alternation edges?]

Note that in Figure 7 that there are two *Loc* vertices. When the traversal schema is traversed at runtime, there will be two separate *Loc* objects instantiated, one by following the *from* edge and another via the *to* edge. An application schema describes the relationships between classes in the application. A traversal schema describes relationships in the runtime *object graph* of the application.

A traversal schema S_t is *legal* with respect to a schema S iff for every edge from vertex $\langle l_f, i_f \rangle$ to vertex $\langle l_t, i_t \rangle$ in S_t has a corresponding edge in S from vertex l_f to vertex l_t .

6.1.1 Traversal Schema Specification with Propagation Patterns

The traversal schema for an edge may be specified using *propagation pattern* notation, which is discussed more extensively in [LX93].

A propagation pattern includes only those elements of a schema which are required to be in the actual traversal schema. This allows the underlying application schema to change while maintaining the validity of the propagation pattern definitions. For example, a propagation pattern to produce the traversal schema of Figure 7 is given in Figure 8.

from	TripRequest, Network
through	\rightarrow *, shortestTrip, *
through	\rightarrow *, reserve, UseList
to	UseList

Figure 8: Propagation pattern to define traversal schema

The propagation pattern in Figure 8 states that the induced traversal schema has “root” vertices of *TripRequest* and *Network* and that there are two edges which must be included in the traversal schema. The first **through** clause says that at least one edge labelled *shortestTrip* must be in the traversal schema, and the second **through** clause requires that at least one edge labelled *reserve*, with a target vertex labelled *UseList*, must be in the traversal schema.

An important note is that the diagram of a traversal schema given in Figure 7 is only one of any number of *possible* traversal schemas which could be induced by the propagation pattern given in Figure 8. Which traversal schema is actually induced by a given propagation pattern is determined by the actual application schema at *propagation time*.

6.2 Edge Code Fragments

The *code* element of an edge definition is a four-tuple,

$$code = (init, start, final, before_list, after_list).$$

Init is a code fragment containing definitions and initializations, in the target language, such as:

```
Vertex* curV = null;
WeightedDiGraph* wdg = new WeightedDiGraph();
```

Any variables defined in the *init* code fragment are guaranteed to be scoped (visible) within any of the other code fragments for this edge. *Start* is a code fragment to be executed before traversal of the schema starts. *Final* is a code fragment to be executed after traversal of the schema has completed. Both *before_list* and *after_list* consist of lists of edge code fragments, which are of the form:

$$edge_fragment = (label, to_class, to_name, fragment)$$

where the optional *label* attribute is the edge name, *to_name* is a name by which the target object may be referenced within the code fragment, and *to_class* is the class of the target of the edge. Edge fragments in an edge's *before_list* are invoked before their respective edges. Likewise, edge fragments in an edge's *after_list* are invoked after their respective edges. If the *label* element of an edge fragment is omitted, the code fragment is executed before or after any edge to the indicated class is traversed.

If there is a name bound to the returned class from the edge signature, there is a default final code fragment which simply returns the named instance. For example, if the signature is:

$$(net : Network \times request : TripRequest) \rightarrow the_trip : Trip$$

the default final fragment is:

```
final: return the_trip;
```

Recall that in Figure 7, there is the following **after** code fragment:

```
after:  $\xrightarrow{reserve}$  u : UseList
       theTrip->set_itinerary(u);
```

This *after* fragment states that whenever an edge labelled *reserve* to an instance of class *UseList* is invoked, the instance of *UseList* is to be locally bound to the name *u* and the line

```
theTrip->set_inventory(u);
```

is to be executed. Note that the **signature** clause has provided an instance of the *Trip* class bound to the name *theTrip*.

For alternation edges, the *over* and *code* elements determine the behavior that should happen when an instance of the target vertex's class is instantiated. For example, if the *start* component of the *code* element of the alternation edge from *Track* to *1Way* has a code fragment, that code fragment will be invoked before the constructor for the class *1Way*.

For repetition edges, the *over* and *code* elements of the edge tuple must be empty.

6.3 Semantics of Schema Traversal

At runtime, when an edge in the application schema is invoked, via a function call, traversal of the edge's *over* schema is initiated.

If the edge's *over* traversal schema is empty, the *init*, *start*, and *final* code fragments of the edge are invoked in order.

Traversal of a non-empty traversal schema starts at the *roots* of the schema – those vertices with no incoming edges. Traversal proceeds by repeatedly choosing an edge which is *prepared* and invoking the edge. An edge is *prepared* if all of its source vertices have been instantiated. If more than one edge is prepared, the edges are chosen in a top-down, left-to-right order, based on the textual representation of the application schema. Traversal halts when there are no more remaining prepared edges.

For example, in Figure 7, assuming the induced traversal schema is as in the picture, the initial set of prepared edges is

$$prepared = \{startTime, from, to, train\}$$

because there is an existing *TripRequest* object at the start of the traversal. If the *startTime* edge is chosen, the prepared set becomes:

$$prepared = \{from, to, train\}$$

If the *from* edge and then the *to* edge are chosen as the next two edges to traverse, the prepared set becomes:

$$prepared = \{train, shortestTrip\}$$

The *shortestTrip* edge has been *prepared* because each of its sources, namely the two *Loc* vertices reached by the *from* and *to* edges, has been instantiated. Once the *shortestTrip* and the *train* edges have been traversed, the *reserve* edge is prepared and is traversed. One of the incoming edges to the reserve edge is a repetition edge from the *TrackList* class, and therefore the reserve edge is repeatedly prepared and traversed, once for each element in the *TrackList* collection. When the *reserve* edge has been finished, the **after**: $\xrightarrow{reserve} u : UseList$ code fragment is invoked. With no prepared edges remaining, the (default) final code fragment is invoked:

```
return the_trip;
```

and the traversal of the *schedule* edge is complete.

If a function edge's *over* traversal schema is empty, and the edge's *code* element is also empty, the edge is implemented as an instance variable.

7 Methodology Highlights and Summary

This methodology combines many concepts which have proven useful in software design and implementation:

- Graph-based
The Demeter research project has developed a successful suite of tools which operate on a graph-based representation of an application's data structures. This approach is incorporated into this methodology.
- Relation and Function-based
The methodology supports a functional view of the applications classes and their interrelationships. More and more complex interrelationships between classes can be built up by using relations to define other relations. The relation edges are incorporated into the graph structure, thus providing an abstract view and set of specification tools for relations. Actual implementation strategies for the relations may be deferred.
- Automatic Graph Traversal
Building on the research on *propagation patterns*, automatic traversal of an object graph is easily defined and incorporated into algorithms. This simplifies the definition of new functions as the composition of a set of existing functions. The traversal of a set of edges in a graph is equivalent to the composition of a set of functions.
- Straightforward Code Generation Strategy
As is (we hope) shown in this paper, abstract schema and edge specifications can be translated into a more traditional language such as C++ in a relatively straightforward manner.

8 Further Work

One of the primary advantages of having all the relationships between classes in a graph is the adaptive software technology based on propagation patterns can take advantage of functional (or derived) relationships between classes. One of the disadvantages of having derived edges in the application schema is that the schema can quickly become cluttered and fully connected. To address this, there needs to be a module system developed, where only the most important edges are actually published and made available to the propagation patterns of other modules.

9 This is the notes.tex file

This file is not to be included in the final version.

Karl- we need a better term than “our methodology” – how about “The Demeter Method”?

Clearly, more references are needed.

9.1 Before and After Edge Code vs. Before and After Vertex Code

As Karl and I discussed, it is arguable that if there is some code which is intended to be invoked only once per instance of a class, rather than every time the instance is arrived at, it may be nice to specify a vertex code fragment. Specifically, if a vertex has multiple incoming edges, specifying

```
*after* *,*,Vertex
  (@ ... code ... @)
```

will cause the code to be invoked after each incoming edge is traversed. In the case where this is *not* the intended behavior, my suggestion is to add another edge from the vertex to itself and include the edge in the propagation pattern. This brings up the concept of *memoized* edges – edges that only do their behavior once and thereafter simply return a previously derived value (or self).

9.2 Virtual Functions, overriding in child classes

The methodology, to be anywhere near complete, **must** support virtual functions which may be overridden by child (or later generations) classes. For example, suppose the abstract *Track* class has a *name* function which the subclass *2Way* wants to override. This could be drawn as in Figure 9, with an edge-to-edge edge indicating the overriding. There would be no reason to actually have any new edges (in addition to the new *name* edge) in the underlying schema graph, because the fact that a child class is overriding a parent class method implementation is easily derivable from the graph. Any time a child class has an edge with the same name as an ancestor class, it is overriding a virtual method.

Having a class with an edge with the same name as an edge of an ancestor class but with the two edges being of different types (function vs. relation or with a different target class) would be illegal.

This is the end of the **notes.tex** file.

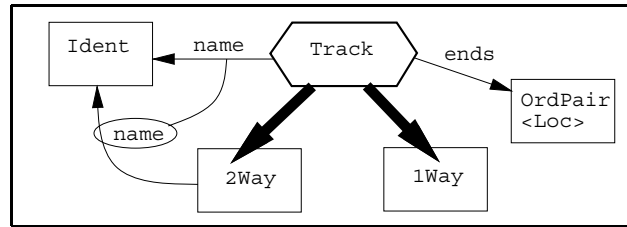


Figure 9: Track Virtual *name* function

References

- [Lie94] Karl J. Lieberherr. *The Art of Growing Adaptive Object-Oriented Software*. PWS-Kent Publishing Company, 1994.
- [LX93] Karl J. Lieberherr and Cun Xiao. Formal Foundations for Object-Oriented Data Modeling. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):462–478, June 1993.

Contents

1	Introduction	2
1.1	Extended Example - Train Scheduling	2
2	Representation of Schemas	2
2.1	Construction Vertices	4
2.2	Relation Edges	4
2.3	Function Edges	5
2.4	Repetition Vertices and Edges	6
2.5	Alternation Vertices and Edges	7
3	Adaptive Programming Using Edges	8
4	Interfaces Induced by Schemas	9
4.1	Interfaces from Relation Edges	9
4.1.1	Interfaces to Relation Objects	10
4.2	Interfaces from Function Edges	10
4.3	Interfaces from Repetition Vertices and Edges	11
4.4	Declarations due to Alternation Vertices and Edges	11
5	Implementation of Relations – Sets, Instance Variables, or Methods	11
6	Edge Definitions – Behavioral Specification	12
6.1	The Traversal Subschema	13
6.1.1	Traversal Schema Specification with Propagation Patterns	14
6.2	Edge Code Fragments	14
6.3	Semantics of Schema Traversal	16
7	Methodology Highlights and Summary	17
8	Further Work	17

9 This is the notes.tex file	18
9.1 Before and After Edge Code vs. Before and After Vertex Code	18
9.2 Virtual Functions, overriding in child classes	18