Abstracting Graph-Based Specifications of Object-Oriented Programs

Ignacio Silva-Lepe
Northeastern University, College of Computer Science

Abstract

This paper presents two contributions towards the automatic maintenance of object-oriented programs. First, a graph-based notation for specifying object-oriented programs is described, referred to as propagation patterns. Second, an algorithm is informally introduced that abstracts this notation from object-oriented programs expressed as call graphs.

Keywords Software maintenance, reverse engineering, object-oriented design and programming, class dictionaries, Law of Demeter.

Introduction

Maintenance of object-oriented programs can be greatly improved if only their essential elements can be expressed using some formal and programming-language independent notation. Using such notation as a vehicle, existing object-oriented programs can be reverse engineered, raising their upkeep to a level where their behavior is easier to comprehend and the necessary modifications and updates are easier to determine.

A characteristic of object-oriented programs that originates one such notation is that they tend to be composed of greater numbers of smaller methods, as opposed to procedural programs, which tend to have fewer number of larger functions or procedures. Wilde and Huitt [10], already document this when they state "There is a general impression that object oriented programs may tend to be structured rather differently than conventional programs. For many tasks very brief methods may be written that simply "pass through" a message to another method with very little processing. Thus a system may consist of a large number of very small modules rather than a relatively smaller number of larger ones." The occurrence of "very brief methods" that perform "very little processing" is a feature that can be exploited. There is a certain amount of communication control work being done in cooperation with the essential details of the algorithm at hand.

A second characteristic of object-oriented programs that crystalizes the development of a notation for their specification is the structural regularity they tend to acquire when they follow the Law of Demeter [5]. Stated briefly, the Law of Demeter (class form) says that inside a method M one should only call methods attached to the class of the current object, to the classes of the immediate parts of the current object, to the classes of the argument objects of M, and to the classes of objects created by M. Classes that communicate in compliance with the law rely a great deal on the part-of hierarchy, thereby displaying predictable patterns of communication.

One should be able to extract the patterns of communication among classes in order to reason about object-oriented programs at a higher level of abstraction. Wilde and Huitt suggest "... that tools for object oriented programming need to address specifically the problem of finding *chains of relationships* rather than simply cross referencing the links in a chain."[10]

This paper presents an algorithm for automatically finding chains of relationships in an object-oriented program. These chains are embodied in *propagation patterns*. Propagation patterns are a formal, graph-based notation for specifying object-oriented programs. That is, the propagation pattern notation is based on a formalization for representing class hierarchies, called *class dictionary graphs* [4].

Propagation patterns expose the essential methods of an object-oriented program by allowing the patterns of communication among the classes that contribute such methods to be generated automatically. Using the connectivity of the class dictionary graphs they have as a substrate, propagation patterns express patterns of communication as *paths* among vertices that represent classes. Paths can also be used to discover patterns of communication to be generated and relevant vertices to be exposed. This way, abstraction of propagation patterns can be automated and existing object-oriented programs can be incorporated into the desired level of maintenance.

Class dictionary graphs

In [4], class dictionary graphs are introduced. Class dictionary graphs are a formal notation for describing class structures that is programming language independent. The class dictionary graph in Fig. 1 describes a class structure to express Scheme programs. A Scheme program contains a list of expressions. An expression is either a variable, an application or a closure. A closure contains an abstraction and an environment. `SchemeProgram`, `ExpressionList`, `Expression` and `Closure` are vertices in the class dictionary graph, also referred to as classes.

`SchemeProgram` is a *construction* class. A construction class is an abstraction of a class definition in a typical statically typed programming language (e.g., C++). Construction vertices are defined in terms of parts. A part is viewed as a high-level concept which might be implemented as a method, not necessarily as an instance variable. In Fig. 1, `expressions` is the only part of `SchemeProgram`. Each construction class defines a set of objects which can be thought

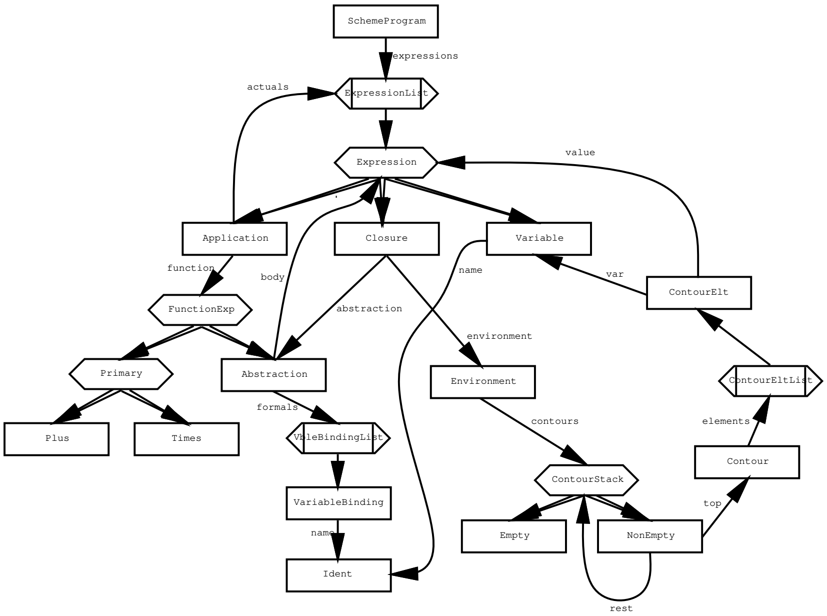

Figure 1: Class dictionary graph for Scheme programs.

of as being elements of the direct product of all objects of the part classes. Part objects do not have to be physical parts, any attribute of an object is a part of it. Thus, the part-of relation is a generalization of the aggregation relation which describes only physical containment. For example, if a Scheme program needs to know about a person as being its developer, we say that the person is a part of the program, although programs are not physically composed of persons.
When modeling an application domain, like that of Scheme programs, it is natural to take the union of sets of objects defined by construction classes. For example, the set of all expressions is the union of the sets of variables, applications and closures. Furthermore, each element in a list of expressions is an element of any of these three sets and no other. Union classes are defined by *alternation* vertices or classes. In Fig. 1, `Application`, `Closure`, and `Variable` are called the alternatives of the alternation class `Expression`. Alternation classes are implemented as abstract classes and their alternatives as subclasses through inheritance. In Fig. 1, `Application`, `Closure`, and `Variable` inherit from `Expression`.
`ExpressionList` is a *repetition* class. Objects of class `ExpressionList` contain collections of `Expression` objects. Repetition classes facilitate expressing container classes. Two important advantages of using repetition classes are (1) the designer need not concern him/herself with the fact that a class belongs to a collection when designing such a class and (2) all the functionality common to container classes, such as iteration, appending, and element count can be abstracted into a single class.
When programming with class dictionary graphs, it is often useful to think about deeply nested objects as objects reachable from their containers. For example, in Fig. 1 an `Environment` object contains a deeply nested `ContourElt` object. This can be recast as saying that `ContourElt` is reachable from `Environment`. Reachability allows the definition of paths of communication between classes.

A graph-based notation for specifying object-oriented programs

One motivation for introducing a notation for expressing object-oriented programs is suggested by Wilde and Huitt [10]: "Understanding of a single line of code may require tracing a chain of method invocations through several different object classes and up and down the object hierarchy to find where the work is really getting done." Propagation patterns allow the specification of the classes where work needs to be done while letting the chains of method invocations to be generated automatically and therefore abstracted away. This way, a programmer can focus on the relevant classes that implement an object-oriented program.
As an example, notice in the class dictionary graph in Fig. 1 that classes `Environment` and `ContourElt` are not directly related, but that `ContourElt` is reachable from `Environment`, as we saw in the previous section. To define these classes as the essential contributors to some algorithm, the path that connects them can be used to generate a pattern of communication automatically. This can be thought of as propagating a message from class `Environment` to class `ContourElt`. Consider looking up the value of a variable in an `Environment`. The essence of the algorithm in this case is to check whether the variable being looked for matches the current variable in the environment, and to assign the corresponding value to the result. Actually, it is also necessary to stop the search when the result has been found. The construct in Fig. 2 describes a program for looking up variables in environments, using the propagation pattern notation.
This propagation pattern specifies only the details necessary to complete the program, in the following steps.

1. The operation to be propagated is specified with the clause
   `*operation*`
   `void lookUp( Variable* v,Expression*& res ).`
2. The keyword `*traverse*` introduces a propagation directive. The source of this propagation directive is given with the clause `*from* Environment`.
3. The target of the propagation directive is given with the clause `*to-stop* ContourElt`. Since class `Envi-`

```
*operation*
    void lookUp( Variable* v,Expression*& res )
*traverse* *from* Environment
            *to-stop* ContourElt
*cut* NonEmpty ContourEltList (@ res != NULL @)
*wrapper* *prefix* ContourElt
(@
  if( this->var->g_equal( v ) )
    res = this->value;
@)
```

Figure 2: Propagation pattern for looking up variables.

ronment is reachable from class ContourElt, and hence Environment $\leadsto$ ContourElt $\leadsto$ Environment $\leadsto$ ContourElt, is a path from Environment to ContourElt, it is conceivable that the propagation could follow this path. However, that is not the intention of this particular program. Thus, the target of the propagation is also specified as a stopping target by the same clause `*to-stop* ContourElt`. If the intention of this program had been to follow the path around the loop, then the target of the propagation would have been given with a clause `*to* ContourElt`.

4. When the result of the search has been found, the program must stop considering any more Contour or ContourElt objects. This could have been done by specifying an actual method for classes NonEmpty and ContourEltList. However, apart from checking for the condition that result is not NULL any more, the code for these two methods looks very much like that of a propagating method. So, the code for these two methods is specified to cut the execution of the propagation using the clause `*cut* NonEmpty ContourEltList (@ res != NULL @)`.

5. Checking whether or not the current variable matches the input and assigning its value to the result if so constitutes the essence of this program. This needs to be specified in a fully user-written code fragment. This specification is done with the clause `*prefix* ContourElt`, introduced by the keyword `*wrapper*`, and followed by the actual code surrounded by "(@" and "@)".

Generating the actual C++ code for the lookUp program involves two steps. First, a subgraph of the class dictionary graph in Fig. 1 is computed that denotes the actual vertices and edges in the class dictionary graph that participate in the propagation. This subgraph is referred to as the *propagation graph* of the propagation pattern. The propagation graph for the lookUp propagation pattern is shown in Fig. 3. Secondly, a separate method is generated for each class in the propagation graph. Each method consists of a collection of call statements, one for each successor of the vertex to which the method is attached in the propagation graph.
In general, a propagation pattern consists of a method interface, a set of propagation directives, a set of cut clauses, and three sets of code fragments. A propagation directive is defined as a 4-tuple composed of the following elements: (1) A non-empty set of source vertices from which the method interface is propagated, (2) A possibly empty set of target vertices where the propagation goes to, (3) A possibly empty set of through edges out of which each path in the propagation graph is required to include at least one, and (4) A possibly empty set of bypassing edges none of which may be included in any path in the propagation graph.

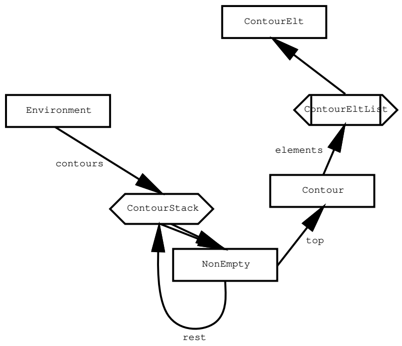

Figure 3: Propagation graph for the `lookUp` propagation pattern.

```
*from* Environment
*bypassing* -> ContourElt,*,*
*to* ContourElt
```

Figure 4: A propagation directive without using `*to-stop*`.

A cut clause consists of a set of classes and a predicate. A cut clause indicates that the execution of the method propagated to the specified classes must be stopped as soon as the given predicate becomes true.
A code fragment designates a piece of code that complements the traversal code generated for a given class. Code fragments are qualified by the class to which the code they complement is generated for. The clause `*prefix* ContourElt` followed by the code surrounded by "(@" and "@)", specifies a prefix code fragment attached to class ContourElt. Code fragments can complement generated code as either `*prefix*` or `*suffix*` fragments.
So far `*from*`, `*to*` and `*to-stop*` vertices have been introduced. However, not every propagation graph that might be useful can be denoted with these three elements. In some situations it may be required that some paths in the propagation graph be bypassed. This situation has already been encountered. The clause `*to-stop* ContourElt` actually specifies that all paths that go past ContourElt be bypassed. The `*to-stop*` keyword is merely syntactic sugar for a combination of `*to*` and `*bypassing*`. For example, the propagation directive in Fig. 2 may actually be written as in Fig. 4, although not as succinctly.
In Fig. 4, the clause `*bypassing* -> ContourElt,*,*` specifies a number of paths that should be eliminated from the propagation graph. Each eliminated path is such that it contains a construction edge with source vertex ContourElt, any label, and any target vertex. Conversely, paths can be specified that are exclusively included in the propagation graph by means of the `*through*` clause.
Propagation patterns can be thought of in terms of paths in a class dictionary graph. Take two classes in a class dictionary graph such that the first class is reachable from the second class. To generate a link of communication automatically between the two classes, the path that connects them could be followed. In fact, there may be more than one path connecting any two classes in a class dictionary graph.
Consider the set of all paths induced by all pairwise combinations of source and target vertices in a propagation pattern ρ. This set is restricted to those paths that contain at least one through edge and that do not contain any bypassing edge in ρ. The resulting set of paths defines a collection of connected subgraphs of the class dictionary graph for which ρ is defined. This collection of subgraphs is merged

into a single graph which becomes the propagation graph of ρ. The meaning of ρ is a collection of method implementations attached to the propagation graph of ρ. The meaning of a propagation pattern can be thought of as a two-tiered denotation. First, the propagation graph of the propagation pattern is computed, then the methods attached to the propagation graph are generated.
A formal introduction to propagation patterns and a definition of their semantics is given in [6].

Abstracting propagation patterns from call graphs

From a maintenance point of view, propagation patterns can help to better understand object-oriented programs when the need arises to enhance or otherwise update them. However, there exist quite a number of object-oriented programs that have been written without using propagation patterns. For all these programs, it would still be desirable to take advantage of propagation patterns to update them. If this is to happen, a mechanism is needed that automatically transforms object-oriented programs into the propagation pattern notation. This way, the maintenance process would only require understanding the essence of a program conveyed by a propagation pattern and what is needed to update the program.
This section presents an algorithm that extracts a propagation pattern from an object-oriented program represented as a call graph. An interesting feature of this algorithm is that it follows very closely the methodology for object-oriented programming that was proposed in the previous section.

Call graphs

The goal of a propagation pattern abstractor is to extract the structural regularity of an object-oriented program and to express this structure as a propagation pattern. As is the case when analyzing an object-oriented program to understand its behavior, a first step is to isolate the call structure of the program. This call structure can then be used to discover patterns of propagation. Tools for extracting call graphs from source code already exist. One such tool is part of the FIELD programming environment [7]. Thus, the input to a propagation pattern abstractor can be assumed to be a call graph expressing the program to be abstracted.
A call graph is a collection of vertices and edges. Each vertex defines a method attached to a class with a specific interface. An edge between two vertices represents a call and is labeled by the name of the method being called and the actual arguments passed. That is, a call edge represents a message being sent, from the body of the method defined by the source vertex to the method defined by the target vertex. Fig. 5 shows a call graph for the `lookUp` program. The vertices in this graph are labeled only with the name of a class for simplicity, but we should be reminded that a vertex in a call graph represents a method attached to a class. Notice that there is an edge from `ContourStack` to `NonEmpty`. This does not mean that there is a call from `lookUp` attached to `ContourStack` to `lookUp` attached to `NonEmpty`. Rather, this means that when `lookUp` attached to `Environment` calls `lookUp` attached to `ContourStack`, one of two things happen. (1) If the `ContourStack` object that receives the message is actually a `NonEmpty` object, then the body of code executed is that of `lookUp` attached to `NonEmpty`. (2) Otherwise, i.e., the receiver is an `Empty` object, the body of code executed is that of `lookUp` attached to `ContourStack`. Expressing dynamic binding of calls to code in this uniform way makes the abstraction process simpler.

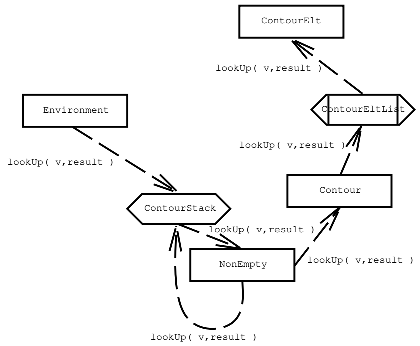

Figure 5: Call graph for `lookUp` program.

Abstraction algorithm

In this subsection, the abstraction algorithm is described informally, highlighting the important issues. The algorithm abstracts a single propagation directive consisting of a single source vertex, a possibly empty set of bypassing edges, a possibly empty set of target vertices, and a possibly empty set of target vertices which stop the propagation (i.e., `*to-stop*` vertices). The inputs to the algorithm are (1) a call graph, (2) the interface definition of the method for which a propagation pattern is being abstracted, and (3) for simplicity, a vertex that will become the source of the propagation. The abstraction algorithm consists of the following steps[1].

1. Transform the call graph into a subgraph of the underlying class dictionary graph.

   As an example, the call graph in Fig. 5 is already a subgraph of the class dictionary graph in Fig. 1. This does not have to be the case necessarily. In this particular case, the program from which the call graph was extracted followed the Law of Demeter. However, call graphs can have any arbitrary structure where any vertex v can have an outgoing call edge to any other vertex w. It could happen that the call graph resulted in a collection of disconnected subgraphs of the class dictionary graph. However, since a propagation pattern can consist of a set of propagation directives, it is legitimate to abstract more than one propagation directive for a disconnected call graph.

2. Identify a propagation directive with a single `*from*` vertex and a set of `*to*` vertices.

   This is done by searching the call graph for vertices that do not send the message under consideration to any of their successors. In our example, such a vertex is `ContourElt`. Special care has to be taken when the search finds a cycle in the graph, e.g., `ContourStack` $\rightarrow$ `NonEmpty` $\rightarrow$ `ContourStack`. Such cycles could contribute `*to*` vertices as well.

3. Compute the propagation graph induced by the identified propagation directive.

   At this point, `*from* Environment *to* ContourElt` has been identified, which denotes a propagation graph ρ. By comparing ρ with Fig. 5, we can identify `*bypassing*` edges.

4. Compare propagation graph with call graph to identify paths to be bypassed.

   In our example, ρ contains a number of paths outgoing from `ContourElt` which are not contained in Fig. 5.

[1] For space reasons, only the main steps of the algorithm are listed. See [9] for a detailed specification of the algorithm.

These paths can be effectively eliminated introducing the clause `*bypassing* -> ContourElt,*,*`. Comparing the propagation graph with the call graph to identify bypassing edges is a sound process since the call graph is a subgraph of the propagation graph.

5. Identify `*to-stop*` vertices. (Optional).

   Finally, the combination
   `*bypassing* -> ContourElt,*,* *to* ContourElt`
   is reduced to simply
   `*to-stop* ContourElt`.

This algorithm currently uses an exact match comparison of the input interface with the interfaces of the methods in the call graph. A relaxation of this assumption is already under way, where the possibility of renaming methods is considered.

Related work

Arango et al [1] describe a transformational theory of maintenance. The abstraction algorithm presented in this paper realizes the abstraction recovery proposed by Arango from the domain level of call graph representation of an object-oriented program to the domain level defined by propagation patterns.
Propagation patterns seem to be unexplored territory. Rumbaugh [8] has proposed an operation propagation mechanism. The most significant difference is that his mechanism is run-time based while propagation patterns are compile-time based and geared towards a new method for developing object-oriented software.
In [2], Harrison and Ossher propose a means of propagating messages between objects that are widely separated in the network of objects that contain them. While in propagation patterns the routing is implicit and given by the underlying class dictionary graph, in structure-bound messages one specifies a default routing for each message type which can be overridden at specific nodes in the network.
Propagation patterns promote the description of behavior at a high-level of abstraction. There exist related constructs which specify behavior with coarser granularity than the class level. Contracts [3] offer a different way of structuring object-oriented programs: Instead of defining the complete interfaces for the classes, the interface information is described in contracts and then injected into the classes.

Conclusions

This paper presents propagation patterns as a graph-based notation to express object-oriented programs. With propagation patterns, the specification of a program can be abbreviated by providing only the essential details that implement the program. Maintainability and understandability of object-oriented programs are improved by using propagation patterns since they offer a way to automatically comply with the Law of Demeter.
The algorithm for abstracting propagation patterns presented in this paper helps to bring existing object-oriented programs into the level of maintainability and understandability made possible by using propagation patterns. Using this algorithm allows a software engineer to move one step forward in the direction of automating software maintenance by focusing on understanding the essence of a program conveyed by a propagation pattern and what is needed to update the program.

Acknowledgements I would like to thank Karl Lieberherr for his guidance, support and feedback. This work was supported in part by IBM Mid-Hudson Valley and in part by the National Science Foundation under grants CCR-9102578 (Software Engineering) and CDA-9015692 (Research Instrumentation).

Author information:
Northeastern University, College of Computer Science
360 Huntington Ave. #237CN
Boston, MA 02115
Phone: (617) 373 3824, Fax: (617) 373 5121
E-mail: nacho@ccs.neu.edu.

References

[1] Guillermo Arango, Ira Baxter, Peter Freeman, and Christopher Pidgeon. TMM: Software Maintenance by Transformation. *IEEE Software*, pages 27–39, May 1986.

[2] William Harrison and Harold Ossher. Structure-bound messages: Separating navigation from processing. Technical Report RC 15539 Revised, IBM T.J. Watson Research Center, Yorktown Heights, NY, October 1991.

[3] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *OOPSLA Conference, Special Issue of SIGPLAN Notices*, pages 169–180, Ottawa, 1990. ACM Press. joint conference ECOOP/OOPSLA.

[4] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. From objects to classes: Algorithms for object-oriented design. *Software Engineering Journal*, 6(4):205–228, July 1991.

[5] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[6] Karl J. Lieberherr, Cun Xiao, and Ignacio Silva-Lepe. An object-oriented implementation of adaptive software. *IEEE Transactions on Software Engineering*, 1993. Accepted for publication.

[7] Steven P. Reiss. Interacting with the field environment. *Software Practice and Experience*, 20(6):89–115, June 1990.

[8] James Rumbaugh. Controlling propagation of operations using attributes on relations. In *OOPSLA Conference, Special Issue of SIGPLAN Notices*, pages 285–297, San Diego, 1988. ACM.

[9] Ignacio Silva-Lepe. *Techniques for Reverse-engineering and Re-engineering into the Object-Oriented Paradigm.* PhD thesis, Northeastern University, 1994. In preparation.

[10] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. In *Conference on Software Maintenance*, pages 162–170, Sorrento, Italy, 1991. IEEE Press.