Adaptive Parameter Passing

Cristina Videira Lopes

Published in Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software (ISOTAS), Kanazawa, Japan. Springer-Verlag LNCS 1049, pages 118-136. 1996.

© Springer-Verlag Berlin Heidelberg 1996.

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Adaptive Parameter Passing

Cristina Videira Lopes *
Northeastern University
College of Computer Science
Boston, MA 02115, USA

Tel: (617) 373 5204; Fax: (617) 373 5121
email: crista@ccs.neu.edu

Submitted to ISOTAS'96

Copyright ©1995 by the author. All rights reserved.

Abstract

Parameter passing is one of the main problems in distributed object-oriented applications. The two simplest solutions - passing objects by global reference and passing complete copies of the object graphs - both have significant drawbacks. Instead, an intermediate amount of copying is often best. The problem is twofold: (i) compilers and operating systems can't automatically make the best decision on how much to copy; (ii) there hasn't been a good mechanism for the programmer to express the intermediate amount of copying that should be done.

The contribution of this paper is to present a new mechanism that allows programmers to express in a natural and succinct way how much of the object graph should be copied to the remote context. The specifications are done using a very simple declarative meta-language – GOOP – that has graphs of classes as domain.

Keywords: distribution, remote invocation, pass-by-copy, open implementation, adaptiveness.

*Supported by the Portuguese Foundation for Research, Science and Technology (JNICT).

1 Introduction

Parameter passing is one of the main problems in distributed object-oriented applications. The co-existence of different address spaces (contexts) that share objects and pass them around in method invocations is a "double-edged sword." On the one hand, sharing objects is one of the basic purposes of distribution, and object-oriented languages offer appealing programming models for developing software applications in general. On the other hand, the existence of a network and the necessity of accessing remote objects makes it impossible to maintain the simplicity of the object models without severe performance losses. Object-oriented operating systems and programming languages have been forced to compromise between these two opposite directions.

To illustrate the problem, consider the following object invocation: $o.m(p)$, where o is the invoked object, m is the invoked method and p is a reference to some other object. When object o is remote to the calling context, what exactly should be passed for parameter p in the remote invocation? On the extreme, two radically different approaches can be made.

The first approach is to copy to the remote context the whole object graph reachable from p. Because objects passed as parameters are highly likely to be invoked by the callee, copying the objects to the remote context is an obvious optimization, so that subsequent invocations to the parameters are performed on the local copies.

The second approach is to not to copy anything. Global references – unique object identifiers that exist above the particular address spaces – are assigned to the parameter objects and only those references are passed. Any subsequent invocations to those parameters result in remote invocations back to the context where the objects execute.

Usually, neither of these two extremes suits the application needs. On the one hand, passing complete copies of the parameter objects may result in replicating the whole data of the application in many different contexts. On the other hand, passing only global references is unfeasible for any practical purposes and does not scale – the proliferation of global references causes an increasing number of cross-context invocations.

For performance and scalability reasons, passing objects by copying them partially is usually a good policy. The problem with it is that the operating system doesn't know what parts of the object will be used in the remote context, that is, the problem arises recursively. By using a pessimistic approach, this may result in copying the whole object graph from one context to another. In many application situations, only a small portion of the object graph is actually used in the remote context.

Therefore, a better solution is to copy only some part of the object graph, and delay the copying of other necessary parts to some point in the future when they are actually accessed. Ideally, however, we would like to send in only one message all the parts that are used in the remote context, and only those parts.

Several distributed programming languages and systems have addressed this problem and have proposed interesting solutions. Emerald [12], for example, provides a set of primitives to define different parameter passing semantics: objects can be passed by reference, by visit or by migration, and the parameter passing mode is defined by the programmer along with the object invocation. When copying is involved, the primitive data of the object gets copied, and references to other objects are promoted to global references. Distributed Smalltalk [2] and COOL [9] also provide explicit object location control associated with parameter passing. More recently, most implementations of CORBA [25] support parameter passing both by global reference (to CORBA objects) and by copy (of non-CORBA objects); when objects are passed by copy, the whole object graph is copied to the remote context, very much like the standard Remote Procedure Call model [4]. Other distributed systems make this issue transparent to the programmer, and implement algorithms that try to optimize parameter passing and object replication in the absence of any semantic knowledge about the applications.

This paper presents a different approach to the problem of parameter passing in distributed object-oriented applications based on the concept of class graphs originally developed in Demeter [16]. We present a new mechanism that allows the programmers to express in a natural and succinct way (i) which objects should not be copied in remote calls; and (ii) for each remote service, and for the parameters that may be copied, how much of the parameters' object graphs should actually be copied to the remote context. The copying specifications are expressed using a very simple graph traversal language, **GOOP** – Graph Object-Oriented Programming.

This paper doesn't address the problems arising from having several replicas of the objects in many address spaces. It solely focuses on the language support for expressing application specific optimizations related to parameter passing.

1.1 The Case Against Well-Intentioned Operating Systems

A lot of research on programming languages and operating systems for distributed applications has pursued the goal of easing the task of the programmers [1] . With this goal in mind, many of these

[1]Throughout this paper, we use the term "operating system" to refer to any kind of run-time support for distributed applications, either a complete operating system or a platform running on top of an operating system.

systems fall on a common mistake: they try to hide the distribution issues from the programmer.

This point has recently been referenced in the literature for clustering of objects. Clustering presents, roughly, the same kinds of problems as parameter passing, since it has been introduced as an optimization for the coexistence of the processes' address spaces and the secondary storage address space. The work presented by Tsangaris and Naughton in [29] suggests that clustering can bring a certain amount of benefits for all applications up to a certain point; at that point any default clustering policy implemented by the operating system will result in good performance for one application, but it will be bad for many others. M. Day in [6] proposes that objects be clustered in groups that may be specific to an application or user and that can be computed at fetch time.

The same argument has also been made, either implicit or explicitly, for synchronization in concurrent and distributed object-oriented applications. Works such as CEiffel [19] and Sina [3], along with the works presented by Frølund and Agha [8], Lopes and Lieberherr [20], and Masuhara et al. [23] suggest that the programmers should be given the chance to define synchronization strategies, rather than being imposed with some default policy implemented by the system.

The work on open implementations and metaobject protocols, by Kiczales [13, 14, 15], generalizes this argument – that hiding implementation can be harmful for performance – and presents an idea for a solution, based on a meta-language for getting at the implementation issues.

All these analyses lead us to believe that with respect to distributed applications, the role of the operating system should be to provide the basic mechanisms for object distribution without imposing any rigid policies. Usually the programmers know important information about how objects in their specific applications should be copied, that no default generic policies can properly address. So, at the level of the operating system, the aim for transparency should not exclude the programmers from defining particular distribution aspects and optimizations. The challenge, then, is to find high-level constructs that are expressive and simple enough for the programmer to use, without making him/her aware of the complexity of their implementations.

1.2 Overview of Our Approach

The goal of our work is to abstract the distribution issues so that they can be controlled by the programmer without making him/her aware of the low-level implementation details. We describe our approach as taking one step back from the state-of-the-art, but going one step up. We go back from powerful operating systems and languages that try to solve parameter passing as transparently as possible to the programmer: we will ask the programmer to supply the key *insights* as to how much

of the objects should be copied; what we need from the operating system is a run time support for remote invocations that takes arbitrary packing/unpacking routines for the parameters. We go up one level of abstraction from classes: GOOP is based on **class graphs** in which the vertices represent user-defined and primitive (integer, string, etc) classes and the edges represent relations between classes. These class graphs contain no more and no less structural information than the set of classes defined in any object-oriented language. However, class graphs express that information in a form that is closer to the design of the application, and that gives the programmers a collective view of the data and relationships in their applications. This collective view is what makes it simple to reason about particular groups of classes and, in particular, to express optimizations for parameter passing.

The remainder of this paper is organized as follows. Section 2 presents the basic concepts of GOOP programs. Next, in section 3, we discuss some particular characteristics of distributed applications and describe the extensions in GOOP graphs as a result of those special characteristics. Both sections 2 and 3 serve as the basis for section 4, in which we describe the solution for adaptive parameter passing in remote invocations, present some examples. Section 5 discusses the issues related to adaptive parameter passing and identifies the open problems. Then, in section 6 we analyze some related work. Finally, section 7 summarizes and concludes this paper.

2 Basics

In this section we describe the basics of **GOOP**, our graph language, without considering any aspects of distribution. The concepts described here are similar to the ones of the Demeter System/C++ [28, 17, 16, 26], but with some important simplifications. GOOP uses a simplified version of the Demeter class graphs that follows roughly the design graphs of OMT [27]. As for the class implementations, GOOP uses C++ methods.

2.1 Class Graphs

The structural aspects of GOOP applications are described by class graphs, where vertices represent classes and edges represent relations between classes. Very briefly, class graphs contain three kinds of vertices and two kinds of edges.

- Vertices:
  - Hexagon: abstract class.
  - Square: concrete class.
  - Ellipse: terminal (primitive) class.

- Edges:
  - Thin arrow from A to B: A has a reference to B. The arrows include a cardinality (the default is *exactly-one*).
  - Thick arrow from B to A: B *is-a* A (subclass relationship).

Fig. 1 shows a class graph for a Library System application. According to this class graph, instances of class Book have references to instances of classes Title, Author, Year and ISBN [2]; instances of class Book also have a collection of references to instances of class Copy (meaning that a book may have more than one copy in the library). Classes Staff and User are subclasses of class Person.

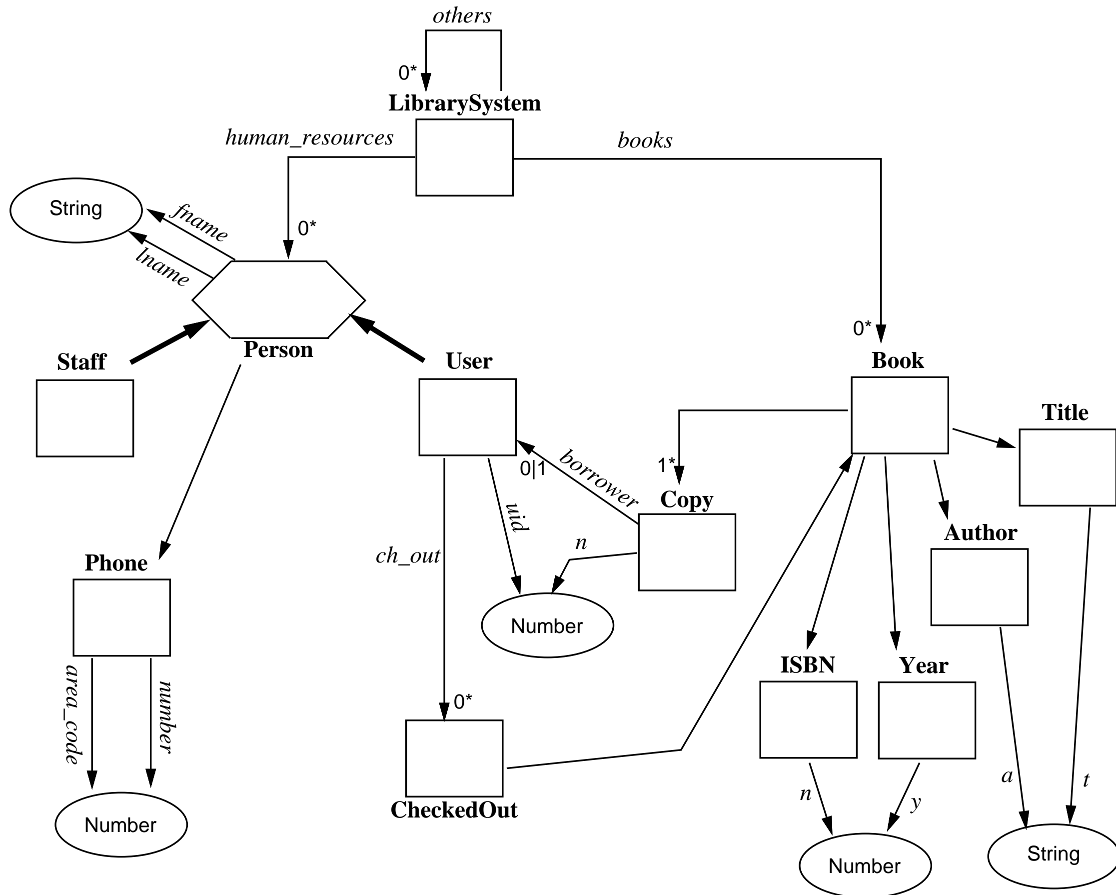

Figure 1: Class graph for a Library System application.

[2] In the Demeter style of object-oriented programming, explicit classes are introduced for many objects that, in other styles of OO programming, would simply use primitive classes. So, for example, in this example, explicit classes are introduced for `author` and `title`, instead of just using the string class. In traditional OOP, the costs of this "additional representation" would outweigh the benefits; the tools of adaptive programming, that automatically generate many graph walks and accesses, cause the advantages to outweigh the costs.

Readers familiar with object-oriented programming can infer the translation of this class graph into partial[3] class definitions. The only important detail concerns the implementation of reference cardinality. For `0*` and `1*` the compiler generates a collection class - a class representing a list of the given element class - as well as a correspondent iterator class - a class that allows iteration on a collection of elements[4].

2.2 Class Implementations

The class implementations in GOOP are written in C++ [7]. Below follows an example an implementation of the operation `find_book`, which, given a key (a string), tries to find a book that contains that key in any of the strings of any of the book objects. This example doesn't introduce anything new – we include it here because we will analyze it as a concrete example throughout the paper.

Only one important method is shown, namely the one in the LibrarySystem class, and we intentionally have kept the implementation as simple as possible - it doesn't even check for recursive calls to libraries. One important remark is that if such book is found, it comes as an output parameter.

```
int LibrarySystem::find_book (/* IN */ String *key, /* OUT */ Book *&book) {
  Book_list_iterator          next_book (*books);
  LibrarySystem_list_iterator next_lib  (*others);
  Book          *bk;
  LibrarySystem *lib;

  while (bk = next_book()) // iterate over books
    if (bk->has_key (key, book) == FOUND) return FOUND;

  while (lib = next_lib())   // iterate over other libraries
    if (lib->find_book (key, book) == FOUND) return FOUND;
  return NOTFOUND;
}
```

3 Class Graphs for Distributed Applications

In building distributed applications, we want to allow the programmer to provide more information to the underlying system, so that certain application-specific optimizations can be made. In particular, the programmer should be able to define which objects serve as *remote entry points* in the global object graph of the application. These objects may be remotely invoked and play a special role in

[3] The structural part.

[4] The programmer must know this. This knowledge is a nasty dependency that can be avoided with further abstraction on the class implementation language. We have intentionally left that abstraction out of this paper, for simplicity sake.

the application both from the programmer's perspective and from the operating system perspective. When these objects are passed as parameters in remote invocations they should not be copied; instead, only a global reference should be passed. (Copying such objects results in replicating servers, and, although this may be a desirable feature in a system, it shouldn't be done through the ordinary mechanism of parameter passing.) Conversely, the programmer should be able to define which objects always execute in the same context, so that invocations to them are always local. This knowledge of the application's distribution "intentions" allows several optimizations, both by the programmer and by the GOOP compiler. For this purpose, GOOP extends the basic class graphs.

3.1 To Be or Not To Be a Server

The capacity to respond to remote invocations is a very important characteristic of the objects and is fundamental for distributed object-oriented applications. Entities that have this characteristic can be thought of as *servers* – using the Client/Server terminology (and for lack of a better word). In fact, they provide the *remote services* corresponding to their interfaces. In distributed object-oriented applications, should all objects provide remote services, transparently, or should there be a distinction between server objects and data objects, as the previous Client/Server systems suggested?

From the operating system's perspective, there is a distinction between objects that may be invoked remotely and those which may not. Server objects introduce an overhead in the system, both in terms of extra code (stubs, marshaling routines, etc) and in terms of performance (remote invocations, complexity of garbage collection, etc).

Making a distinction between server objects and data objects allows certain optimizations at the OS level, because data objects are always invoked locally, can be recycled by some local mechanism, and don't require any extra functionality for the purpose of remote invocations. Many systems support this distinction, either by implementing two different object abstractions (for example, Argus [18]) or by inheritance from different system classes (for example, the CORBA-compliant systems). Having two different object abstractions breaks the uniformity of the object model, and implies that server objects be handled differently from data objects. Defining server characteristics by inheritance or by any other mechanism that preserves uniformity of the objects is a much more elegant solution, since it allows server object references to be passed around in method calls as ordinary data object references.

But it's from the applications' perspective that the distinction is truly important. The design of a distributed application will certainly result in identifying objects which play the role of servers and objects which are less important from a distribution point of view. For example, in the Library

System application shown in Fig. 1, instances of the LibrarySystem class are the obvious candidates for servers, in the sense that they are the *entry points* for all the information in each of the libraries. When an instance of the LibrarySystem class is passed as parameter in a remote invocation, it should not be copied, for that could result in replicating the whole library in the remote context; instead only its global reference should be passed. As for the rest of the classes, it's up to the designers of the distributed version of this application to define whether their instances should provide remote services or not.

In summary: (1) being a server or not is a design decision specific to each class of object in a distributed application; (2) such decision may have important effects on the performance; (3) the replication of server objects is too important to be done through the ordinary mechanism of copying parameters on remote calls; and (4) server objects and data objects should, however, be handled in exactly the same way at the programming level.

3.2 Remote Vertices

Following the previous discussion, we extended the GOOP class graphs with the concept of **remote vertex**. A remote vertex indicates that instances of that class provide remote services, i.e. they are server objects, they accept remote invocations and they are not copied when they are passed as parameters. However, from the programming language point of view they look exactly the same as ordinary data objects: there is no difference in implementation of their methods, they can be instantiated, deleted, passed as parameters in method invocations, etc.

In the Library System example, we choose to define the LibrarySystem class as a remote vertex. The representation of the new graph can be seen in Fig. 2 (only the interesting subgraph is shown).

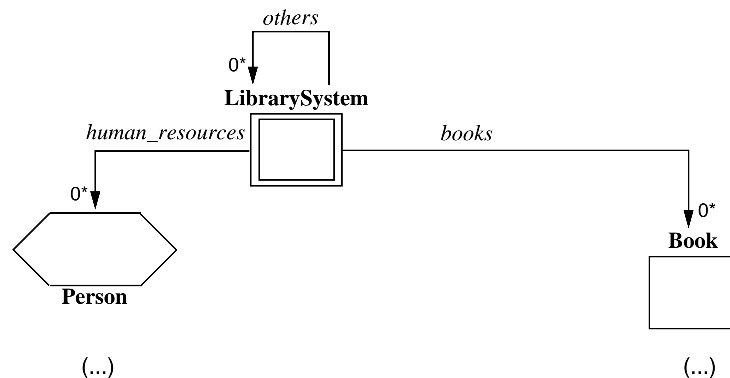

Figure 2: LibrarySystem as remote vertex.

Remote vertices are represented with a double-line border. For the user-defined classes there are no restrictions to being a remote vertex: all vertices in the graph (except the ones which correspond to primitive classes) can be made remote. When an abstract class is made a remote vertex, all its subclasses inherit the ability to provide remote services.

The GOOP compiler generates stubs and marshaling routines for each of the remote service methods. This is the most interesting contribution of this paper and will be explained in the next section.

4 Remote Method Invocations and Adaptive Parameter Passing

This section explains our solution for expressing the amount of copying that should be done for non-server objects that are passed as parameters on remote invocations. We call our approach *adaptive* parameter passing, because it shares some of the ideas of *adaptive software* [16]. GOOP is based on succinct graph specifications – graph directives – that define class subgraphs without referring to all of their vertices and edges. For each class corresponding to a remote vertex in the graph, the programmer defines its interface using GOOP.

4.1 Default

Consider the class graph represented in Fig. 2 (along with Fig. 1) and the class implementation for the operation `find_book` presented in section 2.2. In the implementation of this operation for class LibrarySystem there are method invocations to other objects of class LibrarySystem that may result in remote calls, since class LibrarySystem is defined as a remote vertex.

When a remote call occurs, objects passed as parameters may be copied between contexts; copying implies marshaling the object graph which is reachable from the given object into/from the RPC message, according to the *mode* of the parameters. In the `find_book` example, we want the parameter `key`, of type `String`, to be an input parameter, and the parameter `book`, of type `Book`, to be an output parameter. So, in the interface definition for class LibrarySystem we define:

```
REMOTE LibrarySystem
  OPERATION int find_book (IN String key, OUT Book book)
```

This interface specification instructs the GOOP compiler for the purposes of generation of packing/unpacking routines. Using the above specification, and under a pessimistic approach, the subgraph involved in the marshaling of the second parameter is the one highlighted in Fig. 3.

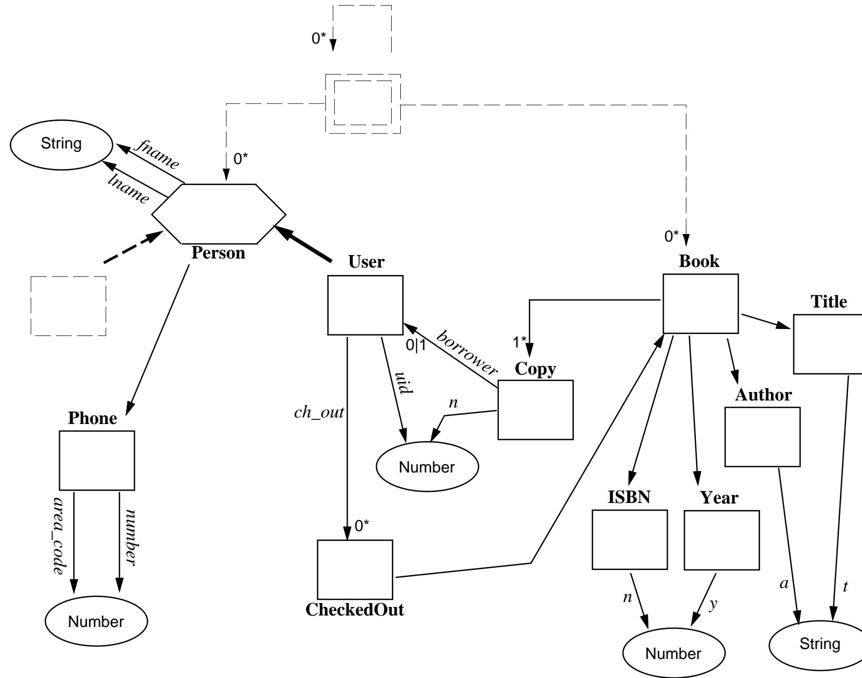

Figure 3: Default subgraph involved in the marshaling of a Book object (solid line).

Only classes Staff and LibrarySystem are excluded, because they are not reachable from Book objects. That is, the Book object eventually returned from the remote call will bring all the other objects which are reachable from it, because of the cycle {Book, Copy, User, CheckedOut, Book}.

Copying all the objects accessible from the parameter object is inefficient, and probably it is not what the programmer wants. The list of book copies, along with the correspondent information about borrowers and all the other books they are borrowing, is a useless overhead for this specific situation. Therefore, this is an obvious case for optimization.

4.2 Graph Directives for Copying Minimization

Fig. 4 presents our solution. In the interface specification, the objects to be copied by the underlying system are defined in terms of graph directives that identify a particular subgraph of the whole class graph. This particular copying directive asks for all paths that start at Book, but only the ones that exclude (bypass) the edge that goes from Book to Copy. Unless otherwise stated, the immediate parts that are of primitive types (String, Number, etc) – that is, the actual data – are always included in the subgraph. The result is the subgraph highlighted in Fig 5 – compare this graph with the default graph of Fig. 3.

```
REMOTE LibrarySystem
  OPERATION int find_book (IN String key, OUT Book book)
    book:
      COPY FROM Book
           BYPASSING -> Book, *, Copy
```

Figure 4: Specification for passing the Book parameter.

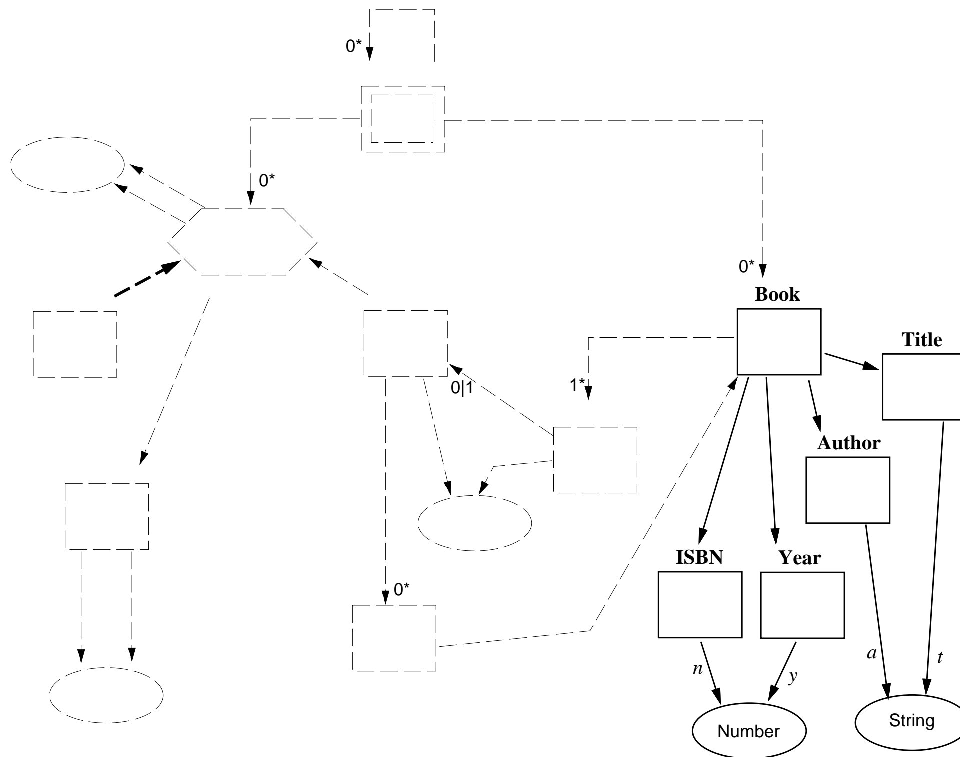

Figure 5: Optimized subgraph involved in the marshaling of the Book object (solid line only).

Our graph language is a subset of the graph language used for propagation patterns in the Demeter system [16, 30]. The formal syntax definition of copy directives is given in appendix A, and the formal semantics is based on the work by Palsberg et al. [26]. Basically, the language contains primitives to define the *source* vertex, the *target* vertices and primitives for path constraints, such as **bypassing**. The meaning of these primitives is as follows. The source vertex and the set of target vertices (if these are specified) define a subgraph of the class graph containing all possible paths from *source* to *target*. The path constraint primitives select only some of those paths: **bypassing** excludes a set of edges; **through** demands for a set of edges; **via** demands for a set of vertices. Note that in the traversal specification of Fig. 4, we need the **bypassing** primitive to avoid the cycle path that goes from Book to Book again, *via* {Copy, User, CheckedOut}.

As another example, suppose that in some generic method M attached to class LibrarySystem we want to pass a parameter of type User, from which we will only access, besides its primitive data, the titles of the books checked out by that user. The subgraph involved is shown in Fig. 6; a possible optimization for this parameter would, then, be:

```
REMOTE LibrarySystem
  OPERATION t M (IN User user)
    user: COPY FROM User
               BYPASSING -> Copy, borrower, User
               TO {Title}
```

As third and final example, we show how adaptive parameter passing can behave as a "filter" for passing objects depending on their types. Consider the graph in Fig. 7, which represents a list of *things*; *things* can be of types A, B or C.

Suppose that in the remote execution of an operation `do(List_of_things *lot)` the programmer knows that only *A_things* will be accessed (for example, Thing-objects will be invoked for method M which is empty for all but for A_Thing-objects). We can optimize the parameter passing by defining the following interface:

```
REMOTE Xpto
  OPERATION void do (IN List_of_things lot)
    lot: COPY FROM List_of_things VIA A_Thing
```

The effect of this optimization traversal is that only the object subgraphs reachable from A_Thing-objects will be copied to the remote context.

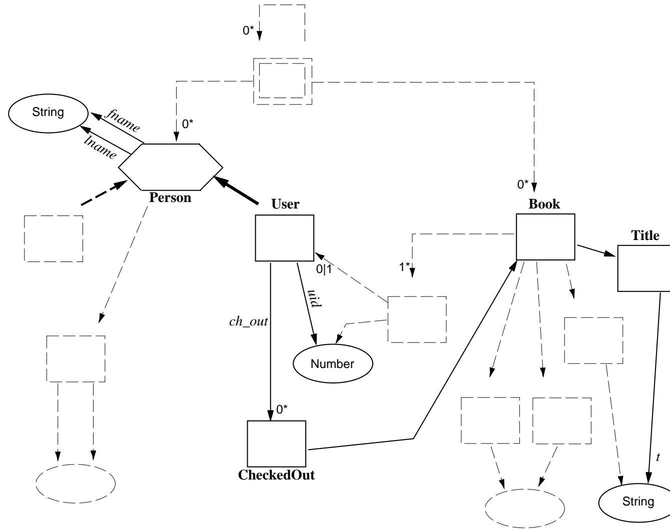

Figure 6: Optimized subgraph involved in the marshaling of the User object.

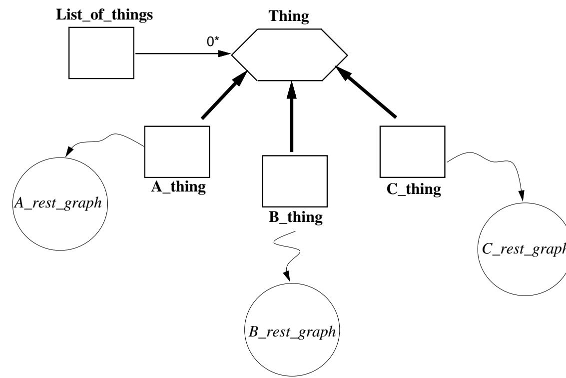

Figure 7: Graph for a list of *things*.

4.3 Marshaling Routines

Based on the graph directives given by the programmer, the compiler generates optimized marshaling code. Below we show the code associated with the GOOP specifications in Fig. 4, in pseudo-code.

```
find_book_params_in (inbuf, key) @ LibrarySystem
     key.marshal (inbuf); // primitive string

find_book_params_out (outbuf, book) @ LibrarySystem
     book.marshal_find_book (outbuf);

marshal_find_book (outbuf) @ Book
     title.marshal (outbuf); // primitive string
```

```
author.marshal (outbuf); // primitive string
year.marshal (outbuf); // primitive number
isbn.marshal (outbuf); // primitive number
```

The pseudo-code shown above is just an example for marshaling the parameters in the `find_book` operation, and it suggests that the marshaling of an object depends on the operation that is being invoked. A more general algorithm for generating marshaling routines follows the outline given below:

For each REMOTE class:
  For each service:
    generate a marshaling routine for the input parameters;
    generate a marshaling routine for the output parameters;
    For each class of the parameters:
      generate a marshaling routine according to the graph directives;

The exact implementation of the packing and unpacking of parameters depends on the underlying support for remote calls, and may have several different flavors. For example, the marshaling of the parameters may be concentrated in only one marshaling routine that accesses all the primitive data or may be distributed in small marshaling routines in the classes involved.

Concerning the marshaling routines, we must clarify one point. Some object-oriented systems associate default packing methods to the classes, and allow the programmers to redefine those methods. The reader may be lead to think that the packing routines generated by the GOOP compiler reimplement that facility. This is correct, to a certain extent, but incomplete. In fact, the GOOP compiler automatically generates specialized packing routines according to the programmer's graph directives; however, the GOOP compiler may generate more than one packing routine for a given class, if objects of that class are passed as parameters in more than one service method. This allows for optimizations that can not be done with only one packing routine per class, and shows, at the implementation level, the expressiveness of GOOP.

4.4 When Copying Hits a Remote Vertex

Since GOOP is a language that defines graph traversals, it may happen that graph directives result in paths that include remote vertices. In this case, the compiler warns the programmer about the fact, and it simply generates the marshaling of a global reference to the server object - that is, no further copying is done from that object.

5 Discussion

In the previous section we have presented the mechanics of our solution. This section makes an analysis of several issues related to this solution.

5.1 Automatic Optimizations

One pertinent question can be raised: if the programmer can identify situations that call for optimizations, could the compiler itself do that? The compiler could make a data flow analysis of the program and compute which parts of the objects are required in the implementations of the methods. We believe that automatic optimizations could be done, but only under certain restrictions and with serious limitations to their effectiveness.

Automatic optimizations would be possible only if the compiler could have access to all modules and libraries of the program, which almost never happens in realistic applications. Consider, for example, the `find_book` operation presented previously, with a book as output parameter; unless the compiler analyses *all* clients of the LibrarySystem, it is impossible to find out exactly which parts of the book should be copied out from the server to the client. A similar argument applies to input parameters – because these may be passed as parameters to other calls in the method's body. And even if such total access could be done, it is not obvious that the compiler could compute the best packing routines, because the parameters may be used in many different ways (e.g. consider a test in the code before accesssing certain parts of the parameter), some of them more frequent than others; "frequency" can only be determined either at run time or by the programmer, who knows a lot more about the application than he is usually allowed to say.

5.2 Encapsulation and Open Implementations

GOOP allows the programmer to define application-oriented optimizations for parameter passing in remote invocations. We assume the optimizations rely on the knowledge that the programmers have about the application they are developing.

This suggests that the principle of encapsulation is broken. In order to do optimizations, the programmers must know how the operations are implemented, which parts of the objects they need. This may be viewed as a serious limitation for the optimization we propose, because it breaks the fundamental abstraction of object-oriented programming – encapsulation.

With respect to this issue, we include our approach in the line of research for open implementations,

as proposed by Kiczales [13] and many other researchers. The abstraction and encapsulation principles of object-oriented programming languages are very appealing for developing applications, but we think that there is a basic discrepancy between those principles and the reality of day-to-day programming, where performance is an issue. Therefore, and for the particular problem of parameter passing in remote invocations, we "open" the implementation, that is, we maintain all the abstractions for the class implementations, but we allow the programmer to perform some optimizations at a higher level of abstraction and according to his/her knowledge of the implementations.

In the solution we propose, the programmer is not confronted with implementation details all the time, and the optimization scheme is made using a very simple graph language to define object structure. We think the simplicity of our solution, and the benefits that it brings in terms of performance, justifies the violation of the encapsulation principle.

5.3 Granularity of the Optimization Mechanism

Our solution is based on *class* graphs. Therefore, the graph directives are specified in terms of classes, that is, the smallest abstraction for the optimization specification is a class, not an object. This means that in our current approach it is impossible to specify that we want to copy only the nth and the mth elements of a list of objects. For the purpose of getting optimal solutions, objects - not classes - are what we need to address.

The current version of GOOP does not provide such level of granularity. However, we think we can improve the current implementation of the language by allowing the programmer to write code wrappers along the traversals of the graph. Such code wrappers, written on the base programming language, may introduce copying strategies that perform some runtime tests before marshaling (or not) the objects.

5.4 Related Problems

At this point we are not addressing the issue of what happens when either the client (for OUT parameters) or the service provider (for IN parameters) need more of the object than what's specified in the graph directive. As far as this discussion goes, we can either assume that the underlying system has some mechanism for fetching the missing parts or impose that the traversal specification be correct in the sense that it must include the object's subgraph that will actually be used in the remote context. Our prototype assumes correctness of the specification, but in a realistic system this assumption would be too restrictive. Fetching objects on demand is not a new idea, and it has been implemented by

several operating systems; adaptive parameter passing should rely on such a system.

Throughout the paper, we intentionally avoided addressing the problem of object replication. This problem is inherent to the work presented here, but it is not introduced by it. In fact, we are simply proposing a language for expressing optimizations for what is already being done - without optimization - by some systems. Replicating objects in different contexts improves the application performance because it reduces the number of remote calls, but it also introduces some additional overhead related to the management of the replicas. Some systems, e.g. CORBA systems, simply ignore the problem by saying that non-CORBA objects - the ones that get copied - are data objects for which the system has no special support. Other systems implement algorithms for replica consistency that assure that each context contains the most recent version of the objects. The language proposed in this paper is relevant for both those approaches.

5.5 Concluding Remarks

As conclusion of this discussion, we present some remarks of a more practical nature.

Scalability We think adaptive parameter passing will scale to very large distributed applications. First of all, we strongly believe that only a small percentage of classes in any distributed application will serve to instantiate objects which are remotely accessible (servers). Second, the complexity introduced by graph directives is minimal: the graph language is extremely simple and it is only used in situations that require optimization.

Prototype Implementation We have developed a small prototype for GOOP which consists of a compiler taking as inputs a class graph, the class implementations and the parameters specifications for the remote services. The result is a C++ program, together with proxy classes and stub routines. All the extra code for remote method invocation is specific for the underlying system. Currently we are using GOOP/RT, a very simple system developed for the purpose of testing these ideas. GOOP/RT is an object server that provides a remote invocation service on C++ objects executing on the same address space; it uses the IK Remote Procedure Call library [21] and runs on top of UNIX.

Performance We intend to test our prototype and obtain performance results on a meaningful set of applications. GOOP allows the specification of optimizations that, we expect, will reduce remote invocation time, garbage collection and that minimize object replication.

6 Related Work

This work is integrated in a much larger project that targets the engineering of distributed object-oriented applications [10]. Therefore, it shares the same conceptual ideas and the same implementation strategy of synchronization patterns [20], namely the fact that the parameter passing scheme – as the synchronization scheme – is defined separately from the class implementation. It also shares, with some simplifications, the basic class graphs and graph algorithms with Demeter [17, 28]. The formal semantics of graph directives is based on the work by Palsberg et al. [26].

The motivation behind this work is the same as the motivation for open implementation [13]. Both lines of research call for application-oriented customizations of the underlying system. Our work consists of a simple declarative meta-language, as suggested by the open implementation community.

In particular, the work in [24] describes a mechanism for object location control using meta-level programming that has some similarities with adaptive parameter passing. In the cited work, method invocations may result in migration of objects between contexts, that is, in moving objects from one context to another, and the policy – not the mechanism – for migration is implemented by meta-objects, not by some default system's algorithm. However, in the reflective systems literature we didn't find any references to the particular problem of parameter passing optimization. We believe that such optimizations can also be made in reflexive systems, but they will have to include some mechanism for describing groups of classes at the meta-level.

The work described in [1] uses Composition Filters to abstract object interactions in a distributed environment through first class objects called *Abstract Communication Types* (ACT's). This work also calls for reflection upon message *send*'s and *receive*'s, so that ACT's be able to implement coordinated behavior, constraint solvers, distributed algorithms, etc. The implementation of the Sina language for distributed computation always passes parameters by copying them to the remote contexts. However, no reference is made in the literature of Composition Filters to any possible optimization of the copying process. In fact, the prototype proved to be very inefficient and they are considering another implementation that uses pass-by-global-reference with optimizations being made by the compiler whenever possible [31].

Our work also has commonalities with work on a very different line of research which consists of the object-oriented analysis and design methodologies. Most of the methodologies presented so far are based on graphs of objects or classes that describe object interactions at the design phase. Examples are the Booch method [5], OMT [27], Objectory [11] and many others. GOOP class graphs can be

seen as design graphs, but we use them directly as program text. Design methodologies usually don't consider any optimizations of the applications. The adaptive parameter passing scheme proposed in this paper handles optimization at a very high level of abstraction, so that it could be considered as part of the design of distributed applications.

7 Conclusions

All the solutions proposed so far for the problem of parameter passing in distributed object-oriented applications delegate to the operating system the responsibility of both the mechanism and the policy of copying the objects between contexts. In the absence of any semantic knowledge about the applications, the best the underlying system can do is either to copy the entire object graph which is reachable from the parameter objects or to implement some generic algorithm that tries to optimize the number of objects to be copied. In any case, the programmer has no control over this process, and there may be severe performance losses due to either copying in excess – parts of the object graph which will never be accessed by the remote context – or copying too little – extra messages will be necessary.

This paper presented a new mechanism that allows programmers to express in a natural and succinct way how much of the parameters graphs should be copied in remote invocations. Our solution makes use of graph-directives written by the programmer to define the amount of objects which are copied between contexts. Graph directives are specified using a simple declarative meta-language – GOOP – that describes subgraphs of classes and instructs the compiler for the purpose of generating specialized marshaling routines.

Acknowledgements:
I would like to thank all members of the Demeter research team at Northeastern University, in particular to Professor Karl Lieberherr, for their valuable comments on applying the concepts of Adaptive Software to distributed applications. I would also like to thank Gregor Kiczales, for his inspiring comments on an earlier version of this paper, and to all members of the OI group at Xerox PARC for brain-storming the base and consequences of this work. Many thanks to the members of the IK research group at INESC, Portugal, for making the IK RPC available to GOOP/RT.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abtracting Object Interactions Using Compositions Filters. In Guerraoui, Nierstrasz, and Riveill, editors, *ECOOP'93 Workshop on Object-Based Distributed Programming*, pages 152–184. Springer-Verlag, 1994.

[2] J. K. Bennet. The Design and Implementation of Distributed Smalltalk. In *OOPSLA '87 Proceedings*, pages 318–330, Orlando, Florida, October 1987.

[3] Lodewijk Bergmans, Mehmet Aksit, Ken Wakita, and Akinori Yonezawa. An object-oriented model for extensible concurrent systems: the composition-filters approach. *Memoranda Informatica*, 92(87), December 1992.

[4] A.D. Birrell and B.J.Nelson. Implementing remote procedure calls. Technical Report CSL-83-7, Xerox, October 1983.

[5] Grady Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings Publishing Company, Inc., 1991.

[6] Mark Day. Object Clusters May Be Better Than Pages. In *4rd Workshop on Workstations Operating Systems*, pages 119–122, Napa, California, October 1993.

[7] Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[8] Svend Frølund and Gul Agha. A language framework for multi-object coordination. In Oscar Nierstrasz, editor, *Lecture Notes in Computer Science*, pages 346–360. ECOOP'93, Springer-Verlag, July 1993.

[9] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov. COOL: Kernel Support for Object-Oriented Environments. In *OOPSLA ECOOP '90, Proceedings*, pages 269–277, Ottawa, Canada, October 1990.

[10] Walter L. Hürsch and Cristina V. Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, USA, February 1995.

[11] Ivar Jacobson, Mangus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach.* ACM Press, Addison-Wesley, 1992.

[12] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, february 1988.

[13] Gregor Kiczales. Towards a new Model of Abstraction in Software Engineering. In *Proceedings of the Int'l Workshop on New Models for Software Archtecture Reflection and Meta-level Architecture*, November 1992.

[14] Gregor Kiczales et al. *Foil for the Workshop on Open Implementation.* 1994. http://www.parc.xerox.com/PARC/spl/eca/oi/workshop-94/foil/main.html

[15] Gregor Kiczales. *Why are black boxes so hard to reuse?* Invited talk, OOPSLA'94. http://www.parc.xerox.com/PARC/spl/eca/oi/gregor-invite.html

[16] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, May 1994.

[17] 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.

[18] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, pages 300–312, March 1988.

[19] Klaus-Peter Löhr. Concurrency annotations for reusable software. *Communications of the ACM*, 36(9):90–101, September 1993.

[20] Cristina V. Lopes and Karl Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In Mario Tokoro, editor, *Lecture Notes in Computer Science*, pages 81–99, Bologna, Italy, July 1994. ECOOP'94, Springer-Verlag.

[21] Cristina Videira Lopes. *IK RPC V3.0: User's and Reference Manuals.* INESC, Lisboa, Portugal, March 1992.

[22] Pattie Maes. Concepts and Experiments in Cmputational Reflexion. In *Proceedings of ACM OOPSLA*, pages 147–155, 1987.

[23] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In Andreas Paepcke, editor, *OOPSLA*, pages 127–144, Vancouver, Canada, October 1992. ACM Press.

[24] Hideaki Okamura and Yutaka Ishikawa. Object Location Control Using Meta-level Programming. In Mario Tokoro, editor, *Lecture Notes in Computer Science*, pages 299–319, Bologna, Italy, July 1994. ECOOP'94, Springer-Verlag.

[25] OMG. *The Common Object Request Broker: architecture and specification*, December 1991.

[26] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 1995.

[27] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[28] Ignacio Silva-Lepe, Walter Hürsch, and Greg Sullivan. A Demeter/C++ Report. *C++ Report, SIGS Publication*, February 1994.

[29] Manolis Tsangaris and Jeffrey Naughton. On the performance of object clustering techniques. In Michael Stonebraker, editor, *ACM SIGMOD Int'l Conference on Management of Data*, pages 144–153. ACM Press, 1992.

[30] Cun Xiao. *Adaptive Software: Automatic Navigation Through Partially Specified Data Structures.* PhD thesis, Northeastern University, 1994.

[31] *Private conversation with members of the Composition Filters research team.*

A Grammar for GOOP Graph Directives

⟨DirectiveExpression⟩	::= **from** ⟨FixedVertex⟩ [⟨PathConstraintExpression⟩]
	{ **via** ⟨VertexSelector⟩ [⟨PathConstraintExpression⟩] }*
	[(**to** ⟨VertexSelector⟩ [⟨PathConstraintExpression⟩]) |
	(**to-stop** ⟨VertexSelector⟩)].
⟨PathConstraintExpression⟩	::= (**through** ⟨EdgePattern⟩ { "," ⟨EdgePattern⟩ }*) &
	(**bypassing** ⟨EdgePattern⟩ { "," ⟨EdgePattern⟩ }*).
⟨VerticesEdgePatterns⟩	::= ⟨VertexSelector⟩ & (⟨EdgePattern⟩ { "," ⟨EdgePattern⟩ }*).
⟨EdgePattern⟩	::= ⟨ConcreteEdgePattern⟩ | ⟨AbstractEdgePattern⟩
	⟨CollectionEdgePattern⟩ | ⟨InheritanceEdgePattern⟩.
⟨ConcreteEdgePattern⟩	::= "->" ⟨VertexSelector⟩ "," ⟨LabelSelector⟩ "," ⟨VertexSelector⟩.
⟨AbstractEdgePattern⟩	::= "=>" ⟨VertexSelector⟩ "," ⟨VertexSelector⟩.
⟨CollectionEdgePattern⟩	::= "~>" ⟨VertexSelector⟩ "," ⟨VertexSelector⟩.
⟨InheritanceEdgePattern⟩	::= ":>" ⟨VertexSelector⟩ "," ⟨VertexSelector⟩.
⟨VertexSelector⟩	::= ⟨AnyVertex⟩ | ⟨FixedVertex⟩ | ⟨VertexSet⟩.
⟨AnyVertex⟩	::= "*".
⟨FixedVertex⟩	::= *identifier*.
⟨VertexSet⟩	::= "{" ⟨FixedVertex⟩ { , ⟨FixedVertex⟩ }* "}".
⟨LabelSelector⟩	::= ⟨AnyEdgeLabel⟩ | ⟨FixedEdgeLabel⟩ | ⟨LabelSet⟩.
⟨AnyEdgeLabel⟩	::= "*".
⟨LabelSet⟩	::= "{" ⟨FixedEdgeLabel⟩ { "," ⟨FixedEdgeLabel⟩ }* "}"
⟨FixedEdgeLabel⟩	::= *identifier*.