A New Approach to Finding Objects in Programs

Panos E. Livadas
Theodore Johnson

Computer and Information Sciences Department
University of Florida, Gainesville, FL 32611

ABSTRACT

Software maintenance is difficult and costly because the maintainer must understand the existing relationships in the maintained code. The maintainer's job can be made considerably easier if the objects in the code (related groups of types, data, and procedures) are identified. In this paper, we discuss methods for identifying objects in programs, and present a new approach that relies on these key features. First, our internal program representation (IPR) lets us make a more precise identification of objects than previous methods allowed. Second, we introduce the idea of receiver-based object identification. Third, we introduce the idea of two-step object identification, which gives the user greater control in precisely identifying objects. Our object finding tool can be used with the other tools our IPR provides to create an integrated software maintenance environment.

1. Introduction

Although conventional programming languages do not directly support object-oriented programming constructs, they do provide several object-like features (such as groupings of related data, abstract data types, and inheritance). It would be very useful to the maintenance programmer to understand the data and function relationships and objects the original designer had in mind. Furthermore, successful maintenance requires a precise knowledge of the data items in the system, the ways these items are created and modified, and their relationships. Identifying objects as well as dependences in procedural code helps to

- understand system design;
- test and debug;
- reengineer the system from a conventional programming language into a object-oriented language;
- avoid degradation of original designs during maintenance;
- facilitate the reuse of existing methods contained in the system.

An object in a conventional programming language can be defined as a collection of routines (functions or procedures), types, and/or data items. Routines implement the methods associated with the object, types structure the data it conceals or processes, and data items represent or point to actual instances of the object class. What is needed is an *identification method, M,* to group objects into a meaningful context of the target program and its real world domain. Let *P* be a program and let $F = \{ f_i : i = 1,2...,l \}$ be the set of its routines. Let $T = \{ t_i : i = 1,2...,m \}$ be the set of its types, and let $D = \{ d_i : i = 1,2...,n \}$ be the set of its data items. We define a *candidate object in P relative to M*, C_M^P, to be a triple

$$C_M^P = (\phi, \tau, \delta)$$

where $\phi \subset F$, $\tau \subset T$, $\delta \subset D$, and *M* is the identification method used in identifying objects.

We use a two-step method of object finding. First, we identify a set of objects via queries to our internal program representation (IPR). Objects found from the IPR are called *primary*

objects. Another set of objects can be found by queries on the primary objects, plus some additional information. These are the *secondary objects*. Because object identification methods are necessarily heuristic, a fixed algorithm for querying the IPR will often provide misleading results. Thus, we decided on a two-step process that gives the user the ability to refine the primary object groupings. In addition, the distinction between primary and secondary object identification permits a clearer description of the identification algorithms.

In section 2, we discuss three primary identification methods: *global-based* ($M=g$), *type-based* ($M=t$), and *receiver-based* ($M=r$); the first two have been proposed in [2]. In section 3, we discuss secondary object identification methods. In section 4, we discuss our internal program representation, and algorithms that return precise primary objects. In section 5, we draw some conclusions.

2. Primary Object Identification Methods

The primary object identification methods that we consider can be grouped into *global-based* ($M=g$), *type-based* ($M=t$), and *receiver-based* ($M=r$) methods. Global and type-based object finding has been discussed in [2]; and receiver-based object finding is a novel approach that is often more accurate than type-based object finding.

We need to make a definition for our discussion of primary object finding. Let x be a variable of type t and let $f \in F$. We say that x is *visible* in f if and only if x is either defined or used in f. If x is visible then x is either a global variable, a local variable (to f), or a formal parameter (to f).

2.1. Global-based Object Identification

Global-based object identification (GBOI) defines a candidate object to be a triple $(\phi,\varnothing,x)$ where $\phi \subset F$ and x is a global variable that is visible to each element of ϕ. Objects identified by this method will be referred to as *globally accessible objects.*

The method in [2] for identifying global objects overlooked two important cases, so their ''objects'' are not precisely identified. First, in languages such as Pascal that permit nested procedures, non-local variables can be regarded as global variables for the purpose of the object finder. Hence, the object finder shouldn't distinguish between global and non-local variables for programs written in these languages. We will use the term *global* to mean both global and non-local variables in the remainder of this paper. Second, in [2] the global-based object finder clusters instances of a global variable with a routine. However, if a variable x is global to a routine f_1, and f_1 calls f_2 with x as the actual parameter, an interpretation should be that x is global to f_2. The algorithm for the object finder described in [2] is unable to find the threads that would be caused by such bindings. As we discuss, our algorithm uses a graph as an underlying structure where bindings, as well as the results of flow-sensitive analysis, are explicitly represented. Therefore, those threads can be identified and traversed.

In the sample program in Table 1, we can distinguish two globally accessible objects. Namely, global variable `gs` is clustered with procedures `push, pop, empty_GS, empty_S` whereas `gq` is clustered with procedures `insert, delete, empty_GQ`. Therefore, the output of the global based object identification algorithm when executed on the program in Table 1 is:

$C_g^P = \{$
$\quad C_g^P[1] = (\{$ `push, pop, empty_GS, empty_S` $\}, \varnothing, \{$`gs`$\}),$
$\quad C_g^P[2] = (\{$ `insert, delete, empty_GQ` $\}, \varnothing, \{$ `gq` $\})\}.$

As an extension, we note that static variables as in C can be considered as global variables of limited scope, since a static variable exists between the invocations of the routines that define it. It is a simple matter to provide the user with the flexibility to treat static variables as globals, and we identify this primary object identification method as the *global-static* ($M=gs$) method.

```
program P;

type elem =
     el_type = array of elem;
               ......
     stack  = record
                    top  : integer;
                    info : el_type;
              end;

     queue  = record
                    front: integer;
                    back : integer;
                    info : el_type;
              end;

     stcque = record
                    part1 : stack;
                    part2 : stack;
              end;

var  gs:  stack; gq : queue;

proc copy (var S1:stcque; var S:stack);
{ insert the top element of S into S1 }

function empty_Q(Q:queue):boolean;
{ return T if Q is empty; otherwise F }

proc insert(x:elem)
{ insert x into (global) queue gq }

proc push_S(var S:stack; var x:elem);
{ push x into S }

function pop_S(var S:stack):elem;
{ pop from S }

function empty_S(S:stack):boolean;
{ return T if S is empty; otherwise F }

proc push(x:elem);
{ push x into a (global) stack gs }

function pop:elem;
{ pop from global stack gs }

function empty_GS:boolean;
empty_GS := empty_S(sg);
{ return T if GS is empty; otherwise F }

proc insert_Q(var Q:queue;x:elem);
{ insert x into Q }

function delete_Q(var Q:queue):elem;
{ insert x into Q }

function empty_GQ:boolean;
{ return T if gq is empty; otherwise F }

function delete:elem;
{ delete from (global) queue gq }
```

Table 1. Program illustrating the object finder.

2.2. Type-based Object Identification

Type-based object identification (TBOI) clusters a routine with the set of types of all its formal parameters and the type of its returned value [2]. For example, each routine $r:(a_1,a_2,.....,a_n)\rightarrow b$ is clustered with the set $\tau = \bigcup_{i=1}^{n}\{a_i\} \cup \{b\}$, where a_i's represent the types of the parameters and b the type of the returned variable (if any). As an example, we applied the *TBOI* algorithm to the program in Table 1. All routines whose type set is τ are clustered together, and the result is:

$C^P_{tp+tr} = \{$

$C^P_{tp+tr}[1] = (\{$ `copy` $\},\{$ `stack, stcque` $\},\varnothing)$,

$C^P_{tp+tr}[2] = (\{$ `push_S, pop_S` $\},\{$ `stack,elem` $\},\varnothing)$,

$C^P_{tp+tr}[3] = (\{$ `insert_Q, delete_Q` $\},\{$ `queue, elem` $\},\varnothing)$,

$C^P_{tp+tr}[4] = (\{$ `empty_Q` $\},\{$ `queue, boolean` $\},\varnothing)$,

$C^P_{tp+tr}[5] = (\{$ `empty_S` $\},\{$ `stack, boolean` $\},\varnothing)$,

$C^P_{tp+tr}[6] = (\{$ `push, pop, insert, delete` $\},\{$ `elem` $\},\varnothing)$,

$C^P_{tp+tr}[7] = (\{$ `empty_GQ, empty_GS` $\},\{$ `boolean` $\},\varnothing)\}$.

It might be useful to cluster a routine with the types of the global and static variables that it accesses as well as the types of its parameters and return value. For example, using the static and global types reduces the number of objects found in program *P* from seven to five:

$C^P_{tp+tr+tg} = \{$
$C^P_{tp+tr+tg}(1) = (\{$ `copy` $\},\{$ `stack, stcque` $\},\varnothing)$,
$C^P_{tp+tr+tg}(2) = (\{$ `push_S, pop_S, push, pop` $\},\{$ `stack,elem` $\},\varnothing)$,
$C^P_{tp+tr+tg}(3) = (\{$ `insert_Q, delete_Q, insert, delete` $\},\{$ `queue, elem` $\},\varnothing)$,
$C^P_{tp+tr+tg}(4) = (\{$ `empty_Q, empty_GQ` $\},\{$ `queue, boolean` $\},\varnothing)$,
$C^P_{tp+tr+tg}(5) = (\{$ `empty_S, empty_GS` $\},\{$ `stack, boolean` $\},\varnothing)\}$.

We observe that perhaps the routine should not be clustered with the type of the return value. For example, the `empty_Q` routine is clustered with the `boolean` type of its return value, but `empty_Q` is naturally an operation on stacks, not booleans.

These considerations lead us to provide three primary *TBOI* methods: parameter type *tp*, global type *tg*, and return value type *tr*. The examples we present in this section use combinations of the three primary object identification methods ($tp+tr+tg$, for example). We discuss methods for combining object identification methods in section 3.

2.3. Receiver-based Object Identification

We define a *receiver parameter type* (or simply *receiver*) of routine *f* to be the type of a parameter that is modified in at least one execution path of *f*. The *receiver-based object identification (RBOI)* clusters a routine with the types of its receivers. One can intuitively realize that a routine might be called with several parameters, but the object associated with the routine contains only a few of these parameters. However, we can often distinguish which of the parameters are the objects because those parameters will be modified. For example, the `push_S` routine takes parameters of types `stack` and `elem`. One typically clusters a push operation with the `stack` type, and in fact the `push_S` routine only modifies the `stack` parameter.

As in type-based object identification, we can apply receiver-based object identification to global and static variables as well. We denote *RBOI* performed on parameters as the *rp* identification method, and *RBOI* performed on globals and statics as the *rg* identification method. For an example of the utility of *RBOI*, we applied a combined $rp+rg$ identification on the program in Table 1:

$C^P_{rp+rg} = \{$
$C^P_{rp+rg}(1) = (\{$ `copy` $\},\{$ `stcque` $\},\varnothing)$,
$C^P_{rp+rg}(2) = (\{$ `push_S, pop_S, push, pop` $\},\{$ `stack` $\},\varnothing)$,
$C^P_{rp+rg}(3) = (\{$ `insert_Q, delete_Q, insert, delete` $\},\{$ `queue` $\},\varnothing)$,
$C^P_{rp+rg}(4) = (\{$ `push, pop, insert, delete` $\},\{$ `stack` $\},\varnothing)\}$.

The stricter requirement of grouping routines with their receiver types returns a finer object set than *TBOI* returns. In our example, *RBOI* returns exactly the objects that one intuitively feels do exist in program *P*. As we show in the section 4, we can identify which fields and subfields of a record a routine modifies, so that we can perform *RBOI* at the level of primitive types. In practice, such low-level information might not be useful.

3. Secondary Object Identification Methods

The object finding methods that we propose in this paper are of necessity heuristic. Since automatic object finding may return spurious or misleading object groupings, the programmer should be provided with flexible object finding tools. We can view the object groupings returned by the primary object identification methods as a database we can use to construct secondary object groupings. Some operations that we can perform are:

- *Selection:* The selection operation lets us perform object finding on a subset of the routines, types, and globals that exist in the program. It is a simple matter to select explicitly named types, routines, and globals, or types, etc., that have some property (for example, select all routines that access global `x`).
- *Union:* The union operation lets us group routines based on mixed criteria. The *RBOI* example used the union of global receiver and parameter receiver object identification to identify its objects.
- *Intersection:* Returning to the example presented in the *RBOI* section, the `insert_Q` and `delete_Q` routines are grouped with the `insert` and `delete` routines. We might feel that `insert_Q` and `delete_Q` are different objects because they access a global variable that the `insert` and `delete` routines do not. We can specify this distinction by grouping routines that both have a `queue` receiver type, but which also access the `Q` global variable.
- *Subtraction:* When grouping routines, we might want to rule out routines that access some particular global or use some particular type. For example, after identifying `insert_Q` and `delete_Q` in an object, we can identify the `insert` and `delete` in an object by requiring that the objects don't access `Q`.
- *Deletion:* The program might contain dependencies that the user feels are to be disregarded during object finding. For example, a simulation program might access a global simulated-time variable in most of its routines, yet these routines naturally belong to difficult objects. We can remove this spurious dependency by deleting references to the simulated-time variable.

Many other queries can be added to the secondary object identification repertoire, once the framework is in place. The secondary object finding methods are similar to relational database queries, and can be implemented using the same algorithms. In the next section, we discuss another query that is very useful for refining the object groupings.

3.1. Least Upper Bound Types

The *TBOI* and *RBOI* identification methods can cluster objects with a large set of types. Intuitively, there are only a few (perhaps one) types in an object. If we are to cluster a routine with a small subset of its types, then intuitively we should cluster the routine with the most complex types [5]. If type x is used to define type y, then we say x is a *part of* y or that y *contains* x. If y contains x, then y is a more complex, or more highly structured, type than x. If T is a set of types, then we define the relation << on T by

$$x << y \text{ if and only if } x \text{ is a part of } y.$$

The << relation forms a Directed Acyclic Graph (DAG) on the types in T. Figure 1 illustrates this relation on the set of types T of the program P in Table 1.

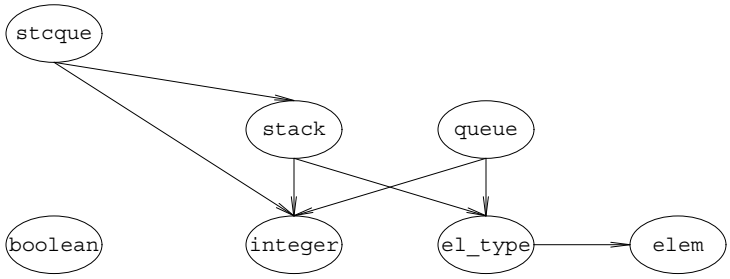

Figure 1. The << relation on the set of types of the program in Table 1.

A *least upper bound* of a set T is an element $l \in T$ with the property that there is no element $e \in T$ such that $l << e$. We will denote the least upper bounds of T by $lub(T)$. As can be seen in Figure 1, the least upper bound of the set of types corresponds to the roots of the DAG. Because << is a partial order, $lub(T)$ will in general contain more than one element. However, $lub(T)$ is likely to be much smaller than T, which will greatly clarify the types that should be grouped with the object. We can further reduce the types that are grouped with the routines by weighting the types (perhaps by their size) and grouping the routines with only the type(s) with the highest weight.

As an example of the utility of using least upper bound types, consider that the first *TBOI* example clustered the `push_s` and `pop_s` routines with the `stack` and `elem` types. Figure 1 shows that $lub(\texttt{stack},\texttt{elem}) = \{\texttt{stack}\}$. Intuitively, it is the `stack` type that forms the object.

4. Algorithms

The first step is to construct the internal program representation, the *System Dependence Graph (SDG)*. The *SDG* models a grammar that permits the following: all primitive (scalar) data types, records, while and for loops, C-like return statements that can be located anywhere in a function, functions that may or not return values (of structured types), nested procedures (as in Pascal) as well as recursive procedures, and differentiation between call-by-reference and call-by-value parameter passing mechanism. We should note that no pointer variables are considered at this time. Furthermore, we must emphasize that the SDG is not constructed for implementing only the algorithms discussed here; it is also the basis for a number of other software maintenance activities -- namely, program slicing, dicing, and ripple analysis [4].

Even though the system dependence graph as well as its construction has been discussed extensively elsewhere ([1], [3], [4]) for the sake of completness we present its highlights. An *SDG* is a directed, labeled multigraph that consists of a *program dependence graph* and a collection of *procedure dependence graphs*. The program dependence graph models the main program and the procedure dependence graphs model the procedure bodies. There is a one-to-one correspondence between procedure bodies and procedure dependence graphs[1]. Each vertex of these graphs represents a program construct such as a declaration, assignment, or control predicate. A collection of additional vertices represent formal parameters, actual parameters, call sites, and entry nodes. The edges represent several kinds of dependences among the vertices and are distinguishable by the labels attached to them. Specifically, we can distinguish three types of dependence edges: control, data, and declaration dependence edges.

In each call site, we create two set of nodes, the *actual_in* and *actual_out* nodes. The former nodes are in a one-to-one correspondence with the actual parameters. The latter are in a one-to-one correspondence with the actual parameters that are passed by reference. By definition, all these nodes are control dependent on the call-site node. Moreover, when a call to a procedure is encountered for the first time, two new sets of nodes, the *formal_in* and *formal_out* nodes are created. They are control dependent to the entry node. It must be clear that the sets of actual_in and formal_in nodes are isomorphic, as are the sets of actual_out and formal_out nodes at the call-site and entry node, respectively, of a procedure. Formal and actual nodes, as well as call-site and entry nodes, are ‘‘linked’’ via edges. Specifically, a *parameter_in* edge is incident from an actual_in node and incident to its corresponding formal_in node; a *parameter_out* edge is incident from the formal_out node and incident to its corresponding actual_out node. A *call edge* is incident from the call-site node to its corresponding entry node.

When a global or a static variable is encountered in a procedure it is introduced as an additional pass by reference parameter to its entry node; therefore, an additional formal_in and formal_out

[1] As discussed later, each aliasing pattern is assumed to give rise to a new procedure body.

nodes are created for this procedure. These nodes are tagged to indicate that they correspond to either global or static variables. Notice that we treat the static variables as global variables of limited scope. The motivation for introducing globals as pass-by-reference parameters is to avoid having flow edges cross procedure boundaries when using or defining global variables [1] whereas static variables are persistent in the sense that they are alive between calls. All procedures that call a procedure directly or those which indirectly use or define global (static) variable(s) are modified to include the global variable(s) as pass-by-reference parameters. The call-sites are also *internally* modified to include the new parameter. For example, when the procedure `example` in Table 2 is encountered and the variable `g` is identified as global[2] while `s` is identified as static, then both variables are treated as pass-by-reference parameters.

```
function example(var y;var x;var z;w;var v):a;
static s;

{
  if s = 0
    { g ← g * g };
  z ← 2 * s;
  s ← g + v;
  x.x1 ← y.y1 + x.x1;
  if y.y2 = x.x1
    { y.y2 ← x.x2
      x.x3 ← x.x2 };
  x.x2 ← 0;
}
```

Table 2. A function abstraction. `x, y, z , w, v, a` and `s` represent the types of the variables whereas `g` is the identifier of a global variable. Subscripted expressions represent the types of subfields of records. For example, x_i is the type of the subfield of a record of type `x`.

Consequently, the function's `example` procedure dependence graph is built as though its prototype had been defined as

```
example(var y;var x;var z;w;var v;var s; var g):a;
```

and the call site would be adjusted accordingly. As mentioned earlier, however, these changes are made on the *SDG* and *not* in the source program, which remains unaltered.

To facilitate precise intraprocedural (and therefore interprocedural) data dependence analysis, when a passed parameter is of a non-scalar type such as a record we ''decompose'' the record into the set of its primitive components. At the same time we allocate a set of nodes whose cardinality is the same as the former set so that they can be properly represented. For example, referring to the program in Table 2, when the call to procedure `example` is encountered we create the new variables $x.x_1$, $x.x_2$, $x.x_3$, $y.y_1$, and $y.y_2$ besides the instances of the records `x` and `y` (Figure 2). Decomposition not only permits precise data dependence calculations but enables us to identify subrecords as receivers, as we demonstrate shortly.

Functions in both Pascal and C require special attention. For both languages, we introduce two new types of edges. The first, the *affect_param edge*, denotes the dependence between an actual parameter and the return value of the function (i.e., if a parameter influences the return value of the function, then an affect_param edge is incident from the actual_in node corresponding to the parameter of its call-site). The second, the *return_link edge*, is an interprocedural dependence edge which denotes the dependence between the return nodes in the function and the corresponding call-site.

---

[2] This is accomplished by a simple ''look-up'' in the scoped declaration table that is maintained during the construction of the *SDG*.

A *transitive dependence edge* exists from an actual_in node to an actual_out node if the formal_out node corresponding to the latter node is intraslice-path-reachable from the formal_in node (that corresponds to the formal_out node). Notice that this is equivalent to saying that the intraprocedural slice of the PDG at the formal_out node *contains* the formal_in node. The collection of the transitive dependence, affect_param dependence, and return_link dependence edges at each call site is defined as the procedure's *summary* information. Figure 2 illustrates the summary information of the function `example`.

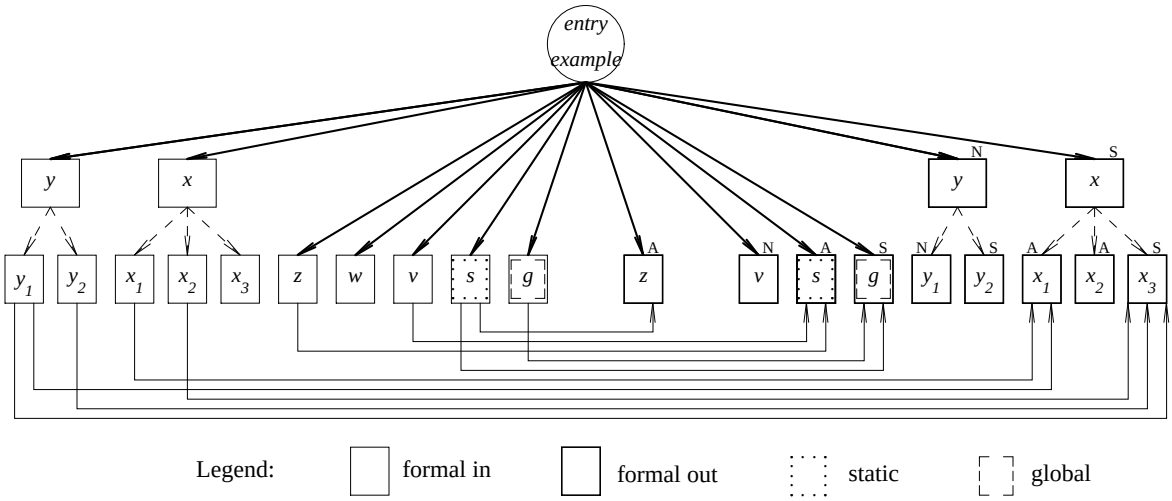

Figure 2. The portion of the program dependence graph corresponding to the function `example` and its summary information. Bold solid arrows represent control dependences; dashed arrows the "<<" relation; and solid arrows represent transitive edges. Other edges such as affect_param are not shown.

During the construction of the *SDG* we can recognize and therefore classify a formal_out node as either an N, S, or A node. The first case occurs when the variable is passed to the procedure and is never modified, the second when it is sometimes modified, and the third when is always modified. For example, in Figure 2 the formal_out nodes have been classified. Notice that for our purposes a formal out node corresponding to a record is classified as either an S node or an N node. The classification is equal to the logical or of its subordinate fields with A and S nodes corresponding to true and N nodes to false.

4.1. Aliasing

The problem of aliasing (given our grammar) arises when a call to a procedure is made with two or more parameters that share the same memory[3] location. A technique that will identify direct and indirect aliasing and the dependency analysis of procedures in the presence of aliasing has been discussed elsewhere ([1], [4]). We have created a separate copy of the procedure for each unique alias configuration and solve the procedure taking into account the variables that are bound to the same memory location. If there is a call with an identical alias configuration which has already been solved, we do not make another copy of the procedure and solve it; we merely reflect the summary information from the procedure that was previously solved. Since aliased objects are of the same type, we must examine the entry node of the procedure that corresponds to

[3] Global variables are already treated as parameters.

the most ‘‘alias-free’’ configuration of the procedure. This means that the object finder uses the entry node that corresponds to the minimum number of aliased parameters.

4.2. Unknown Procedures

One of the advantages of using the SDG is that it can represent a program even if the bodies of the procedures are unknown, such as library procedures. The only information that is required is the data dependences between the parameters and the return type of the procedure. A simple way to capture these procedure dependences is to write a ‘‘stub’’ routine and include it in another file. The user’s source file and this file are then merged; the SDG is then constructed on the merged file. For example, to model a procedure whose first and second parameters are passed by value and whose third parameter is passed by reference and is also dependent on both the first two parameters and on itself, a stub procedure could be written as shown in Table 3.

```
procedure example(a, b: integer; var c:integer);
{
 c ← a + b + c;
}
```

Table 3. Modeling unknown procedures.

It is not necessary for the code of the ‘‘stub’’ procedure to look anything like the real procedure we are trying to model. In the case of unknown procedures, we are only interested in the data dependences, so the the summary information can be derived.

4.3. Identifying *M*-based Objects

Algorithms that identify global-, type- and receiver-based objects using the system dependence graph are discussed in the following paragraphs.

We have two cases of determining global-based objects. In the first case, no global variable is passed as a parameter to another procedure. Therefore, we only examine the summary information at each entry node and cluster all tagged, formal-in parameters with their procedure. In the second case, when globals are passed to a procedure as a parameter, some additional work is required. To visualize the problem more clearly assume a call sequence such as `A`→`B(y)`→`C(z)`; let `g` be a global variable defined or used in `A` that is passed as a parameter to `B` and then in turn to `C`. Then `g` can not be ‘‘threaded’’ based only on its name; a more powerful technique is required. For this reason we introduce a new edge called *global_thread* edge which is incident from the formal_in node to the corresponding actual_in node of the parameter. Our example demonstrates that there would be a global-thread edge from the formal_in node corresponding to `g` (in `A`) with the actual_in node corresponding to `y` (in `B`). Also notice that the latter node must be tagged to indicate that it corresponds to a global variable, and also linked with its corresponding formal_in node via a global_thread edge. Hence, clustering a global variable with the procedure(s) with which it is defined (or used) directly or indirectly is accomplished by a simple traversal of the path that has as its starting vertex the node representing the global variable in question and consists of only global_thread edges.

Determination of type-based objects consists of clustering all non-tagged, formal_in parameters and the return type of the most alias-free configuration of the procedure. Determining whether or not a function returns a value is done by examining to see if a return_link edge is incident to some call site associated with the procedure in question; in this case, the return_link edge is followed ‘‘backwards’’ to determine the type of the returned value. We note that our method permits one to cluster the type of a global variable to a procedure given our internal program representation.

Finally, to determine receiver-based objects, the formal_out nodes of the most alias-free configuration of a procedure are examined. If the formal_out node is sometimes or always modified, then the variable has been changed in the procedure, and therefore it becomes a receiver.

5. Conclusions and Future Work

Identifying object-like features in code written in conventional programming languages is a useful step in applying object-oriented methods to the reengineering and maintenance of old code and in porting programs written in procedural languages to object-oriented languages. We have shown the shortcomings of the global-and type-based methods suggested in [2] and [5], proposed a more precise method based on receiver types, and presented a flexible method for secondary object finding. We have implemented the object finder and the dependence analysis tool for a large subset of ANSI C and Pascal. We are currently exploring new methods to handle pointer variables in the context of the internal representation that we employ.

6. References

[1] S. Horwitz, T. Reps and D. Binkley, *Interprocedural slicing using dependence graphs*, ACM TOPLAS, January 1990.

[2] S.S. Liu, N. Wilde, *''Identifying objects in a conventional procedural language''*, in Proceedings of the Conference on Software Maintenance, November 1990.

[3] P.E. Livadas, S. Croll and P.K. Roy, *''Towards an Integrated Software Maintenance Environment''*, in Proceedings of the 1st Symposium on Software Engineering Research Forum, Tampa, Florida, November 1991.

[4] P.E. Livadas, S. Croll, *''Program Slicing''*, SERC-TR-55F, Software Engineering Research Center, Computer and Information Sciences Department, University of Florida, October 1992.

[5] R.M. Ogando, *''An Approach to Software System Modularization Based on Data and Type Binding''*, PhD thesis, University of Florida, Gainesville, Florida, 1991.