Incremental Call Graph Reanalysis for Object-Oriented Software Maintenance*

Amie L. Souter and Lori L. Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
{souter, pollock}@cis.udel.edu

Abstract

A program's call graph is an essential underlying structure for performing the various interprocedural analyses used in software development tools for object-oriented software systems. For interactive software development tools and software maintenance activities, the call graph needs to remain fairly precise and be updated quickly in response to software changes. This paper presents incremental algorithms for updating a call graph that has been initially constructed using the Cartesian Product Algorithm, which computes a highly precise call graph in the presence of dynamically dispatched message sends. Templates are exploited to reduce unnecessary reanalysis as software component changes occur. The preliminary empirical results from our implementation within a Java environment are encouraging. Significant time savings were observed for the incremental algorithm in comparison to an exhaustive analysis, with no loss in precision.

1. Introduction

As bandwidth increases on the Internet, web sites are becoming more application oriented. Users are starting to use the web to run applications such as word processors, image filters, graphics software, virtual desktops, and gaming. One primary characteristic of web based applications is the use of the object-oriented paradigm, because of its important software engineering advantages. In particular, Java, because of its portability, is increasingly dominating the implementation of these applications. Java is also popular due to the large available collection of libraries and the opportunities for code reuse from its object-oriented development.

Because object-oriented software is typically characterized by many methods, with a large number of them being quite small, sophisticated program analysis for compiler optimization, software testing, debugging, semantic program browsers and other software development tools perform *interprocedural* analysis. Interprocedural analysis involves program call graph construction and computation of desired information, typically data flow information, by traversing the call graph. A call graph represents the calling relationship between methods in a program. A node in the graph represents a method and a directed edge between two nodes establishes the calling relationship between the caller and callee. While a sound call graph can be computed quickly, a more precise call graph leads to more useful information for the client software tool. However, dynamically dispatched message sends in object-oriented software make more precise call graph construction nontrivial. The complexities have led to many call graph construction algorithms with varying levels of precision and expense [20, 13, 15, 1, 5, 8, 6, 14, 7, 18, 23, 24].

As an application changes in composition with components being added, deleted, replaced, and updated during the software maintenance life cycle, the call graph for the application changes, which can impact the validity of the interprocedurally gathered information. Given the expense of computing a call graph with sufficient precision, our goal has been to develop an incremental call graph construction algorithm with acceptable cost for use with interactive software tools. In addition to avoiding an expensive exhaustive call graph construction, an incremental call graph algorithm also easily indicates the potentially affected regions to begin incremental reanalysis of the gathered data flow information. It is particularly desirable to reuse results from prior analysis in the context of object-oriented software involving shared libraries.

The incremental call graph construction problem is nontrivial for object-oriented software for the same reasons that complicate exhaustive call graph construction - dynamically dispatched message sends and the interdependent roles of type information and call graphs. Furthermore, in an object-oriented system, edits other than call site changes can af-

*This work was supported in part by NSF EIA-9806525 and NSF EIA-9870370.

fect the call graph structure because type information is changed, which impacts the information known at polymorphic call sites. In this paper, we present incremental algorithms to update a call graph that has been initially constructed using the Cartesian Product Algorithm developed by Agesen [1]. The exhaustive algorithm constructs a very precise call graph based on the concept of templates. The properties of templates make this approach to call graph construction an ideal candidate for efficient incremental call graph construction, in addition to retaining a highly precise call graph during software maintenance.

We begin with a brief overview of exhaustive call graph analysis in the context of object-oriented software. The exhaustive Cartesian Product Algorithm is described with emphasis on the notion of templates. After describing the incremental call graph problem in more depth, we present our incremental algorithms for deleting and adding a call site. Some empirical results comparing the incremental to the exhaustive approach are presented. We conclude with an overview of related work on incremental data flow analysis and demand driven analysis, and then conclusions and future work.

2. Exhaustive Call Graph Analysis

At compile time, a conservative call graph can always be built, by including a set of edges for each call site, such that edges from nodes representing the caller points to nodes representing potential callee methods based on a conservative static analysis. However, in the presence of dynamically dispatched message sends, a call graph built in this fashion will most likely contain extra edges that do not reflect any dynamic calls. Interprocedural analysis can be performed on such a call graph, but the analysis suffers from imprecision due to the extraneous information propagated along unnecessary call graph edges [7]. In addition, because a call graph is used to determine the set of reachable methods in a program, many software engineering applications also benefit from time savings due to identifying and analyzing only reachable methods.

In order to produce a more precise call graph for an object-oriented program, the set of potential receiver classes at each call site must be more precisely computed, which requires that precise type information be determined. Polymorphism makes determining precise type information difficult because an object can potentially be bound to different types throughout the execution of a program. Specifically, interprocedural class analysis, which is closely related to the call graph construction problem, is used to determine precise type information throughout a program. By simultaneously performing interprocedural analysis and building a call graph, a more precise call graph can be constructed and more precise receiver class information can be computed.

2.1. Overview of Approaches

Many approaches for exhaustively computing precise call graphs for object-oriented software have been developed [20, 13, 15, 1, 5, 8, 6, 14, 7, 18, 23, 24]. Grove et al. formulated a framework that reveals the relationship between the precision of call graph construction algorithms and the overall impact that various call graph construction algorithms have on interprocedural analyses [10]. In particular, they define the *ideal* call graph for a given program to be the greatest lower bound over all call graphs corresponding to a particular execution of the program. In general, it is impossible to compute the ideal call graph directly, because there may be an infinite number of possible program executions, but call graphs can be ordered with respect to the ideal call graph. A call graph is *sound* if it safely approximates any program execution. A call graph that represents only a particular execution of a program is said to be *optimistic*. Thus, the ideal call graph is the most optimistic sound call graph. And, a call graph is sound iff it is equal to or more conservative than the ideal call graph.

Most call graph construction algorithms are in the sound region of the call graph domain, because they result in sound call graphs. However, the algorithms vary in where they fall on the lattice of the sound domain. Very conservative, sound approximations fall at the bottom of the lattice, for example, call graphs created by Bacon and Sweeney's rapid type analysis algorithm [4] or class hierarchy analysis [6] developed by Dean et al. Precise call graph construction algorithms fall closer to ideal; for example, Shivers 0-CFA [20] and Agesen's Cartesian Product Algorithm CPA [1]. The Grove et al. study includes algorithms from [20, 15, 8, 4, 14].

The Grove et al. study also shows that interprocedural analyses can lead to effective optimizations with the use of a precise call graph, but these call graph construction algorithms are often costly and do not scale well to large programs. Recently, call graph construction algorithms have been developed that scale better for large programs. In particular, Tip and Palsberg have shown precise call graph construction algorithms to scale to a 325,000 line Java program [24]. Rountev et al. have also recently developed a new bounded inclusion constraint points-to analysis, which can be used for call graph construction [18]. The Sable Group at McGill University is also developing scalable call graph algorithms [23]. Their work improves on rapid type analysis [4] and class hierarchy analysis [6] by incorporating a flow insensitive type analysis. They have developed two variations of their technique, namely variable-type analysis and declared-type analysis.

2.2. Cartesian Product Algorithm

The Cartesian Product Algorithm (CPA) developed by Agesen is a call graph construction algorithm that addresses dynamically dispatched message sends with the advantages of simplicity, precision, generality, and efficiency due to its avoidance of iteration and redundant analysis [1]. The key insight of the CPA algorithm is to partition each call site into a set of templates.

In object-oriented languages, each method includes a (possibly empty) set of formal parameters. Each formal parameter is typed, allowing objects of certain types to be passed as actual parameters. Due to inheritance and dynamic binding, each formal parameter is potentially mapped to a set of types. A set of templates for a call site is created by computing the Cartesian product of the types of the actual arguments. The class hierarchy is used to compute the set of types for each formal parameter of the method. Then reaching definitions are used to calculate the types that reach each actual parameter at a given call site; such an approach is described by Agesen in [2]. A different template is created for each combination of reaching types for each actual parameter at a call site, based on the computed Cartesian product.

A template can be viewed as an instance of a method, with exact type information for each formal parameter, and representing a particular combination of actual parameter types. Each template then has a singleton set of types for each formal parameter, either equal to the declared type or narrower than the declared type. In other words, a template is an exact type signature of a method. A single method potentially can be mapped to many templates. Because exact type information is available for each template at a particular call site, iteration is not necessary during type inference (and call graph construction). Instead, CPA achieves precision because it creates monomorphic templates, which provide exact type information.

As an example, Figure 1 shows a simple method, methodX, with one call site, y = x.methodZ(a). The types reaching the receiver argument, x, and actual parameter, a, form the sets of types {Type1, Type2} and {Type3, Type4}, respectively. By calculating the Cartesian product over the set of reaching types to the receiver and actual parameter, the set of templates shown in Figure 1 is generated at this call site.

The caller-callee relationship between templates can be used to construct a precise call graph. A graph connecting the templates can be constructed similar to a call graph. We call this the template call graph. The template call graph has unique properties not found in a traditional call graph. The graph represents the caller-callee relationship in the same manner, but traditionally each edge of a call graph has no type information associated with it. In the template call

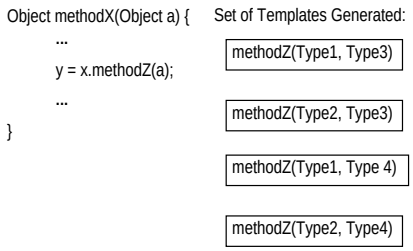

Figure 1. CPA template example.

graph, type information is implicitly associated with each edge. Any template maintaining a caller-callee relationship is also maintaining the fact that a particular exact set of types is flowing from caller to callee. In section 4, we will see how these properties are exploited in incremental call graph construction.

The relationship between the template call graph and a traditional call graph is relatively simple. Where the template call graph's nodes represent templates, the traditional call graph's nodes represent methods. In order to transform the template call graph to a traditional precise call graph, a simple pass is performed over the template call graph that determines the most precise actual method for each template. For each actual method in the program there is a mapping to the set of templates associated with it. For each call site in the template there is an edge to a template, which corresponds to an actual method. Therefore, by traversing each call site in each template we can build a precise call graph that uses the templates to determine the exact set of methods at each call site.

3. Incremental Analysis Problem

The goal of an incremental algorithm is to reuse results from previous analysis in order to update the desired information (in this case, a program call graph) in an attempt to perform an amount of work proportional to the original change's impact, thus avoiding unnecessary reanalysis. Incremental algorithms are particularly beneficial when the exhaustive computation is an expensive operation to perform in response to program maintenance changes. The largest improvements in reanalysis time are typically in response to changes with small impact applied to a large software system that would require significant redundant recomputation if performed exhaustively.

An underlying premise of most incremental algorithms is that an initial exhaustive analysis has been performed such that the information to be kept up-to-date is as precise and correct as desired prior to any program changes. In the case of incremental call graph construction, we assume that a precise call graph was exhaustively computed for the program, and the incremental algorithm is to be invoked upon

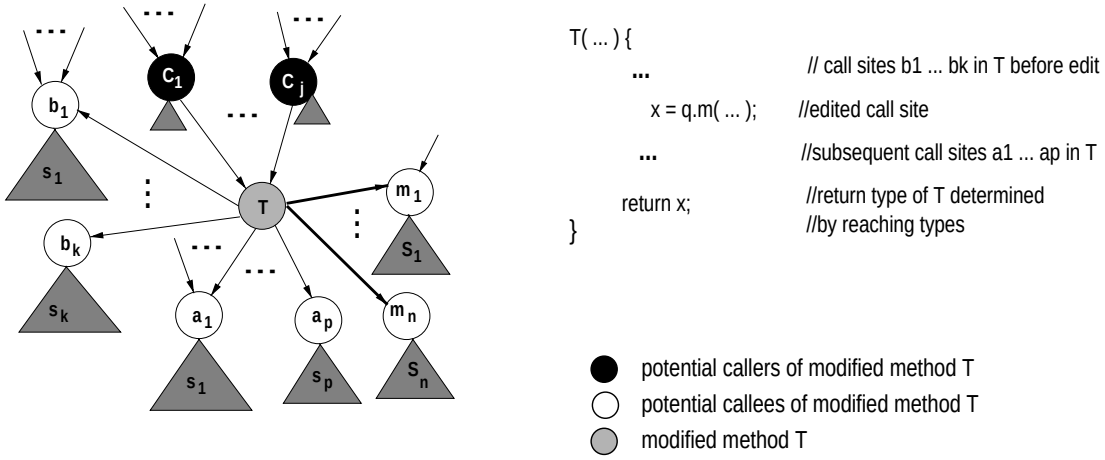

Figure 2. Incremental call graph analysis requirements.

program changes.

For object-oriented languages, the call graph can be affected by any edit that directly changes the calling relationship between methods or by an edit that impacts the type information at call sites. Edits that can *directly* change calling relationships include:

- insert or delete a call site
- replace a call site
- change an actual parameter expression
- change the receiver argument
- change the method name

Changes to an actual parameter or receiver argument alter the type signature, which changes the set of potential callees at the modified call site. The last three changes can be handled as subsets of replacing a call site, which can be handled as a deletion followed by an insertion. Edits that can *indirectly* impact reaching type information at call sites and thus affect the call graph include:

- insert or delete a definition of a variable that reaches the use of a variable appearing in an actual parameter
- insert or delete a flow graph edge
- insert or delete a call site that returns a value used later
- change the variable storing the return value at a call site
- insert or delete a method
- insert or delete an inheritance link

Program changes that do not have an impact on the call graph until later changes are made include inserting or deleting a class (requires class instantiation, inheritance link updates, or insertion/deletion of methods) and inserting or deleting a declaration of an instance or global variable (requires insertion/deletion of definitions or uses).

In this paper, we describe incremental algorithms for the direct changes. Because all direct changes can be handled as a subcase of either insertion or deletion of a call site, we focus on these edits. Indirect changes can be handled as changes in the reaching type data flow information. We discuss how to address changes in reaching type information as part of the incremental algorithms for direct changes. Handling these indirect changes is a subproblem of handling direct changes, and thus not described separately.

For concreteness, consider deleting the call site CS: x = q.m(...) from the skeletal code segment in Figure 2, which contains an abstract illustration of the part of the call graph related to this call site. The required updates to maintain soundness and precision throughout incremental analysis in response to deleting a call site are outlined as follows:

1. The edges in the call graph that correspond to the deleted call site must be deleted. In the figure, the bolded edges from the node labeled T to nodes m_1 to m_n correspond to the deleted call site. There is a set of edges due to the fact that the call site was determined to be polymorphic with a set of receiver objects bound to q, each with its own method m, which we call m_1 through m_n.
2. The type information propagated from the modified method T, through the actual parameters to the potential callees at the edited call site, m_1 ... m_n, must be updated. The type information propagating from other callers to each m_i must be examined, and then the call

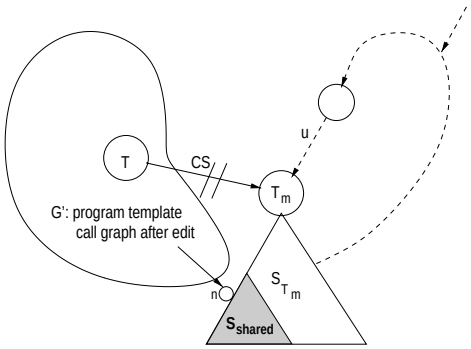

Figure 3. Template call graph update scenario.

subgraph, S_i, of each m_i should be updated to reflect the updated type information.

3. The set of return types bound to x from the set of all the callee's m_1 ... m_n, must be used to compute the changes in reaching types(both newly reaching and killed types) of x immediately after CS in T.

4. The type information propagating to any potential subsequent callees, a_1 to a_p, in the modified method T, and those callee's subgraphs, s_1 to s_p, must be updated.

5. The return type set of the modified method, T, must be updated if necessary to correspond to the updated types associated with the deleted call site.

6. The callers of T, C_1 to C_j, (and their subsequent callees and callers) must also be updated to reflect any changes in the return type information propagating back to each caller of T.

4. Incremental CPA

In Figures 4 and 5, we provide a detailed algorithm for deleting a call site. We also provide a brief overview of inserting a call site, which has very similar steps.

4.1. Maintaining Precision with Templates

The incremental CPA algorithm presented here avoids any reanalysis of the deleted template itself and its subgraph by exploiting a key property of templates and the template call graph. Unlike a call graph or control flow graph, a template call graph has the property that at a given node, every incoming edge carries exactly the same type information. When an edge is deleted, either the sink node becomes unreachable, or the remaining edge(s) retain the exact same type information coming into the node as before the edit. When the sink node becomes unreachable, we perform no further update to the now unreachable node and its subgraph so that when the template is needed later, its subgraph is immediately up-to-date with the desired precision.

Consider Figure 3, where the edge labeled CS is deleted. First, we look at the case of T_m becoming unreachable, and the possibility of the subgraph, S_{T_m}, rooted at the sink template, T_m, sharing part of its subgraph, say S_{shared}, with the still reachable portion of the modified program's template call graph G' after the edit. For a subgraph to be shared, G' must contain some edge that leads into at least one node, n, of S_{shared}. So, does S_{shared} need to be updated by propagating changes from T_m so that G' remains sound and precise? No, because node n carries the same information as carried from the subgraph rooted at T_m, and thus the shared subgraph, S_{shared}, retains soundness and precision without any update.

Next, consider the possibility of at least one additional incoming edge to T_m, say u, remaining after the edit. The question becomes – Does S_{T_m} need to be updated? There are two cases to consider: (1) The edge u is entering T_m because u and T_m are contained in a strongly connected region with the deleted edge CS being the only entry. In this case, S_{T_m} is unreachable after the edit, and we perform no update so that S_{T_m} will remain precise when reused. (2) T_m remains reachable from G' after the edit, due to u. Then S_{T_m} remains sound and precise because u carries the same type information as the deleted edge. Otherwise, S_{T_m} could be reachable due to another incoming edge into a strongly connected region that contains both u and T_m, and u is dependent somehow on the type information from CS to T_m. However, if S_{T_m} remains reachable from G', then there must be some other entry into the strongly connected region that carries the same type information as CS, and thus u and S_{T_m} remain reachable from G', and G' remains sound and precise without further update. In summary, whether T_m remains reachable or unreachable from the modified program's template call graph, no update is needed for S_{T_m} to retain precision and soundness in G'.

4.2. Algorithm to Delete a Call Site

The delete_call_site algorithm shown in Figure 4 updates the template call graph, G, after the deletion of call site CS has occurred in method M. The algorithm begins by deleting the edges in the template call graph associated with the edited call site CS. Since multiple templates could be mapped to the modified method, the delete_edge_and_save_template function is called for each of these templates to delete the call site edge and save

Algorithm Delete_Call_Site(G, M, CS)

Input: G: template call graph
M: method being edited
CS: call site x = q.m(...)being deleted
Output: Updated template call graph G'

```
foreach template node T of method M do
   foreach template C ∈ calleesof(T,CS) do                          //steps 1 and 2
      delete_edge_and_save_template(T, C);
   delete_types_after(CS, x) = ⋃ gen_types(C,x) ∀ C;                //step 3
   add_types_after(CS, x) = ⋃ reaching_types_out(p,x), ∀ p ∈ pred(CS), ∀ C;
   propagate_update(CS, x, delete_types_after, add_types_after, T); //step 4
   Let del_types(T) = set of deleted reaching types at return from T; //step 5
   Let add_types(T) = set of added reaching types at return from T;
endfor
foreach template CT ∈ caller(M)do                                   //step 6
   foreach C ∈ callsite(CT) to callee M do
      Let y = variable storing return value at C;
      delete_types_after(C,y) = ⋃ del_types(T), ∀ templates T of M;
      add_types_after(C,y) = ⋃ add_types(T), ∀ templates T of M;
      propagate_update(C, y, delete_types_after, add_types_after, CT);
   endfor
endfor
```

Figure 4. Algorithm Delete_Call_Site.

the template (and its subgraph) associated with the sink node of the deleted edge. This process corresponds to steps 1 and 2 described in the previous section.

Next, the set of types bound to the return variable at the call site CS must be computed in order to update any subsequent call sites where the return variable types reach. There are actually two set of types to calculate, 1) the set of types that CS generated, delete_types_after, and 2) the set of types that now reach past the deleted call site CS, add_types_after. This corresponds to step 3 in section 3. These sets are propagated to subsequent call sites in the template T by calling propagate_update. Then, step 5 computes the set of deleted types and added types that reach the return of each template T associated with the modified method M.

The second foreach nest corresponds to step 6, which propagates the two sets of types back to the callers of the modified method M. This step is performed by first finding all the callers of the modified method M, and then for each call site C in the caller that calls M, we calculate the types that need to be deleted and added at call sites after C. The sets delete_types_after and add_types_after are calculated by unioning the del_types and add_types for each template corresponding to method M. Algorithm propagate_update is executed to propagate and update the set of reaching types to subsequent calls in the caller of M.

The algorithm propagate_update in Figure 5 propagates and updates the set of reaching types for each of the subsequent calls after a call site with modified type information. The input to the algorithm is the call site, CS, affecting the reaching type information in the current template T, the variable x whose type information has changed, the set of types deleted immediately after the call site CS, and the set of types added immediately after the call site CS. The algorithm performs two general tasks, 1) propagate and update the set of deleted reaching types, and 2) propagate and update the set of added reaching types. The algorithm processes the deleted types first by performing an intraprocedural incremental data flow update in T based on the set of deleted reaching types at call site CS. Any algorithm for intraprocedural incremental data flow update of reaching types (or definitions) could be used[27, 16, 19] The incremental data flow analyzer returns a worklist that contains the call sites in T that were influenced by the deleted reaching types at CS.

Each call site in the worklist is analyzed as follows. If the set of deleted reaching types at the call site C has a nonempty intersection with any of the parameter types of a template CT for C, then the edge to the template CT is deleted, and the template CT is saved similar to deleting

the call site C at that point. After each template CT is analyzed for the call site C, the impact of the changed return types at the call site C are identified in delete_types_after, and propagate_update is recursively called to update those changes.

After the worklist from incrementally updating deleted reaching type information in template T is processed, a worklist for added reaching type information is created through a separate incremental data flow update within T. The call sites on this worklist are processed similar to the first worklist, except new templates are added with new edges to them as newly reaching type information is determined at a given call site. The function compute_new_templates computes the set temp_list of new templates at a given call site, and analyze_and_add_new_templates performs an abbreviated version of the exhaustive algorithm for creating a template call graph starting at the new template, and reusing templates. The changes in return type information from new templates are propagated similar to the return type information changes from deleted templates, by a recursive call to propagate_update.

4.3. Inserting a Call Site

Updating the call graph after the insertion of a call site is similar to deleting a call site. Overall, the steps from the delete_call_site algorithm only need minor modifications to be adapted for inserting a call site. We need to insert an edge in the template call graph from each caller template associated with the modified method to the potentially new callee. In order to correctly add the edge, we must determine the types for each parameter of the inserted call site. We do this by simply calculating reaching types to each parameter, and then computing the Cartesian product over the types to generate the set of templates that represent the callee of the call site. The callee template may already exist, and then we can reuse it, or we may have to create the template and then analyze the template from that point forward. The fact that the callee template may potentially exist provides a savings in time for the incremental update. After generating the template and the edges, we must analyze the return type and determine how the new return type affects the call sites after the newly inserted call site. We again determine the added types and the deleted types, which are computed slightly different than the delete_call_site algorithm. In deleting a call site, there were potentially a set of types that were no longer killed by the deletion. When we insert a call site, we are potentially killing a set of types that previously did not get killed. Next we proceed with steps 3-6 for deleting a call site, which remain the same when inserting a call site.

4.4. Time, Space, and Precision

The incremental CPA algorithm focuses the reanalysis effort on those portions of the template call graph which remain reachable after the program edit, and are potentially affected by the changed return type of the edited call site. The reanalysis is directed by changes in reaching type information in subsequent callees and callers of the modified method. Due to the key property of the template call graph in which all incoming edges to a given template carry the same exact type information, there is no update needed for the subgraph shared by the reachable program template call graph and the deleted template. Reuse of templates between different versions of the program also speeds up incremental call graph construction.

Precision is not lost by the incremental CPA algorithm over multiple edits because both added and deleted reaching type information are propagated and associated call graph changes made. A faster incremental update could be performed if only type changes that preserve monotonicity of type inference were propagated, but precision would be lost during incremental analysis, especially over multiple edits, resulting in a call graph with an increasing number of unnecessary edges.

While incremental call graph construction based on CPA benefits from the properties of templates and their easy reuse, a natural concern is the additional space needed for the template call graph in comparison to a call graph. The number of templates per method is dependent on the polymorphism and the number of arguments for a given method in the target program. Agesen [2] suggests that a reasonable sized template call graph computation would result from a program having at most 3-5 possible types reaching each receiver or actual argument of a given method, on average. In our experiences, we have not come across programs that are "too polymorphic", or in Agesen's terms, *megamorphic*. However, in order to address the potential for scaling to these kinds of programs in the future, we are investigating the use of summaries.

5. Empirical Results

To demonstrate the potential benefits of our incremental call graph construction algorithm, we have implemented the delete_call_site algorithm and experimented with edits to a sizable Java program. We use the FLEX compiler infrastructure [12] from MIT in order to process Java bytecode and build the initial exhaustive data structures such as the call graph and class hierarchy. We have extended their implementation in order to incrementally update the call graph to reflect edits (call site deletions) to the code.

We used an example program from the Soot framework [25] as our case study. Table 5 shows some of the char-

Algorithm Propagate_Update(CS, v, del_set, add_set, T) //step 4

Input: CS: call site affecting reaching type information in T
v: variable for which reaching type information is changing
del_set: set of deleted reaching types immediately after CS
add_set: set of added reaching types immediately after CS
T: template associated with the method containing call site CS
Output: Updated propagation of reaching type and call graph changes

```
worklist = incre_df_update(del_set, CS, T);
foreach call site C in worklist do
    y = variable storing return value of C;
    foreach template CT ∈ calleesof(T,C) do
        Let del_types = set of reaching types deleted at C;
        Let parameter_types(CT) = set of types in type signature of CT;
        if(del_types ⋂ parameter_types != null) then
            delete_edge_and_save_template(T, CT);
    endfor
    delete_types_after(C,y) = (⋃ old_gen_types(CT,y) ∀ CT) - (⋃ new_gen_types(CT,y) ∀ CT);
    propagate_update(C,y,delete_types_after,{},T);
endfor //end of processing deleted reaching types

worklist = incre_df_update(add_set, CS, T);
foreach call site C in worklist do
    y = variable storing return value of C;
    foreach template CT ∈ calleesof(T,C) do
        Let add_types = set of reaching types added at C;
        Let parameter_types(CT) = set of types in type signature of CT;
        if(add_types - parameter_types !=null ) then
            temp_list = compute_new_templates(add_types - parameter_types);
            analyze_and_add_new_templates(temp_list);
    endfor
    add_types_after(C,y) = (⋃ new_gen_types(CT,y) ∀ CT)- (⋃ old_gen_types(CT,y) ∀ CT);
    propagate_update(C,y,{},add_types_after,T);
endfor //end of processing added reaching types
```

Figure 5. Algorithm Propagate_Update.

acteristics of the case study program. The first three rows show general size characteristics of the program: number of classes, methods, and bytecode instructions that compose the program. The next two lines characterize the associated call graph, including the number of singleton and polymorphic call sites, which sum to give the size of the call graph in terms of edges. Finally, the average number of templates mapped to a method indicate the size of the template call graph in relation to the call graph.

The goal of our case study was to provide a preliminary estimate of the time savings of our incremental approach in response to deleting call sites with different characteristics. We executed the incremental algorithm for 40 polymorphic call sites, each with some of the following characteristics: (1) callee with no return type or only a primitive return type, thus not requiring any update of reaching type information after the edited call site, (2) callee with an object being returned, (3) callees mapped to various number of templates, and (4) callers mapped to various number of templates. We measured the time to execute the incremental algorithm, the number of template call graph edges deleted in response to a single deleted call site, and the time to execute the exhaustive algorithm.

The experimental results from our case study indicate a potentially large savings over the exhaustive analysis. The average percent of time savings over the exhaustive analysis for edits that required a large reanalysis was 5%. The effort needed for incremental reanalysis is clearly dependent

No. of Classes:	726
No. of Methods:	4466
No. of Instructions:	75090
No. of Singleton Call Sites:	16284
No. of Polymorphic Call Sites:	1393
Ave no. of Templates:	12.6

Table 1. Benchmark characteristics.

on the number of templates associated with the modified method and call site edge. Deleted call sites with no return types and having only one template associated with the caller or callee obviously provided a significant time savings over an exhaustive analysis. While we feel that our results provide insight into the potential of an incremental approach, we also believe that the time savings could be less with programs that exhibit different characteristics than the one we used for our case study. In particular, edits in the context of programs that require the changed types to be propagated more extensively will require more analysis.

6. Related Work

In this section, we describe related work in incremental and demand driven analysis. Both of these approaches have been developed with the goal of reducing program analysis time by analyzing only a limited portion of a program. Techniques for incremental analysis rely on an initial exhaustive computation, and focus on reducing the effort to recompute in response to program changes. Alternatively, demand driven techniques avoid an exhaustive program analysis, and instead compute data flow facts in response to a query. Data flow facts are computed over a selective region of the program that influence the data flow information at the program point of the query.

6.1. Incremental Analysis

Iterative-based, interval-based, and hybrid incremental data flow analysis algorithms were first developed for intraprocedural data flow analysis [27, 16, 19]. Yur, Ryder, and Landi developed interprocedural algorithms for updating procedure side effect information as well as pointer aliasing information [26]. Pointer aliasing analysis is key to gaining precise analysis of C programs in which pointers play a large role, and also in object-oriented programs in which variables are references to objects. Yur's incremental version of the Landi-Ryder flow and context sensitive pointer aliasing algorithm for C programs is shown to provide a 6-fold speedup over the exhaustive algorithm, when a statement deletion is performed. About 75% of the tests had the same solution as the exhaustive analysis [26].

Very little work has been done in incremental analysis within the object-oriented context, where inheritance and dynamically dispatched message sends need to be addressed. Harrold et al. demonstrate how to reuse information from testing parent classes to "incrementally" test subclasses by focusing the subclass testing on new or affected inherited attributes [11]. Recently, Vivien and Rinard studied incrementalized pointer and escape analysis, where they analyze only those parts of the program that may deliver useful results. More closely related to our work are the efforts of Grove [10] and Agesen [2]. In his dissertation on effective interprocedural optimization of object-oriented languages, Grove describes an overall framework for incremental interprocedural reanalysis and recompilation in the Vortex programming environment [10]. In particular, he discusses the requirements for incremental call graph construction based on the call graph framework that he developed; however, he does not provide any specific details or algorithms. Agesen's dissertation on concrete type inference for object-oriented programs also includes a brief section discussing incremental CPA [2]. He describes and characterizes the possible effects of different program changes on the inferred types. No algorithms for performing the required updates are presented.

6.2. Demand Driven Analysis

Reps et al. model the demand driven interprocedural data flow analysis problem as a graph reachability problem [17]. The graph representation is called an exploded supergraph, which is an expansion of the program's control flow graph to include an explicit graphical representation of each node's flow function. This approach suffers from the large size of the exploded supergraph when attempting to scale to large programs. Duesterwald and Soffa developed a demand driven interprocedural data flow analysis framework based on a query system [9]. Queries are raised by the application either manually through an interactive software tool or automatically generated by software such as an optimizing compiler. To answer the query, the query is propagated in the reverse direction of the original exhaustive analysis until all necessary points in the program are covered to determine the response to the query. A partial data flow analysis is performed based on the propagation of queries.

Agrawal developed a demand driven call graph construction algorithm that solves the problem of computing the set of procedures potentially called at a particular call site without computing this information for every call site [3]. Class hierarchy analysis is first performed on the entire program to create an initial conservative call graph, followed by demand driven analysis that calculates more precise type information used to refine the set of possible call graph edges

for the call site of interest. The algorithm updates information for a single call site, and thus is suggested for scenarios in which precise call graph information is only needed at certain call sites on a demand basis, such as program slicing or performing optimization during just-in-time compilation.

In contrast, incremental call graph construction is more appropriate when a precise call graph has already been built for such purposes as interprocedural program optimization or software testing, and a precise call graph is desired as program changes occur. It should also be noted that a demand driven call graph construction algorithm is only concerned with narrowing type sets, whereas incremental call graph construction in response to program edits must handle the possibility of both narrowing and widening reaching type sets in order to maintain precision.

7. Conclusions and Future Work

To our knowledge, this paper presents the first detailed algorithms for incremental call graph construction that are both applicable to languages with dynamically dispatched message sends and retain high precision during software maintenance. The reuse of results from prior call graph analysis is particularly desirable for directing incremental reanalysis of interprocedural information used in interactive software development tools. The capability to perform incremental analysis for object-oriented software development tools is increasingly important as object-oriented software with heavy use of shared libraries becomes the paradigm of preference to encourage software reuse, and Java continues to play a key role in web and mobile code development.

We are currently developing regression testing techniques based on the OMEN approach to structural testing of object-oriented programs [21]. The incremental CPA algorithm will play a central role in these regression testing techniques, which are an integral part of our TATOO testing and analysis tool [22] for object-oriented software. We are also designing a more extensive evaluation study of our incremental CPA algorithm.

References

[1] O. Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings of ECOOP*, 1995.

[2] O. Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford, 1996.

[3] G. Agrawal. Demand-driven call graph construction. In *Proceedings of the Compiler Construction Conference*, 2000.

[4] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of OOPSLA*, 1996.

[5] C. Chambers and D. Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of PLDI*, 1990.

[6] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of ECOOP*, August 1995.

[7] G. DeFouw, D. Grove, and C. Chambers. Fast Interprocedural Class Analysis. In *Proceedings of POPL*, 1998.

[8] A. Diwan, J. E. B. Moss, and K.McKinley. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. In *Proceedings of OOOPSLA*, 1996.

[9] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM TOPLAS*, 19(6):992–1030, Nov. 1997.

[10] D. Grove. *Effective Interprocedural Optimization of Object-Oriented Languages*. PhD thesis, Univ of Washington, 1998.

[11] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrich. Incremental Testing of Object-oriented Class Structures. In *Proceedings of ICSE*, May 1992.

[12] M. Rinard et. al. FLEX. www.flex-compiler.lcs.mit.edu, 2000.

[13] J. Palsberg and M. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of OOPSLA*, 1991.

[14] H. Pande and B. Ryder. Static Type Determination for C++. In *Proceedings of USENIX C++ Technical Conference*, 1994.

[15] J. Plevyak and A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proceedings of OOPSLA*, 1994.

[16] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12):1537–1549, Dec. 1989.

[17] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Data Flow Analysis via Graph Reachability. In *Proceedings of Principles of Programming Languages*, 1995.

[18] A. Rountev, A. Milanova, and B. G. Ryder. Points-to Analysis for Java Based on Annotated Constraints. In *Proceedings of OOPSLA*, 2001.

[19] B. Ryder and M. Paull. Incremental Data-Flow Analysis. *ACM TOPLAS*, 10(1):1–50, January 1988.

[20] O. Shivers. *Control Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, 1991.

[21] A. Souter and L. Pollock. OMEN: A Strategy for Testing Object-Oriented Software. In *Proceedings of ISSTA*, 2000.

[22] A. Souter, T. Wong, S. Shindo, and L. Pollock. TATOO: Testing and Analysis Tool for Object-Oriented Software. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, 2001.

[23] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Valle-Rai, P. Lam, E. Gagnon, and C. Godin. Practical Virtual Method Call Resolution for Java . In *Proceedings of OOPSLA*, 2000.

[24] F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms . In *Proceedings of OOPSLA*, 2000.

[25] R. Valle-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework . In *Proceedings of CASCON*, Sept. 1999.

[26] J.-S. Yur, B. G. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of ICSE*, 1999.

[27] F. Zadeck. Incremental Data Flow Analysis in a Structured Program Editor. In *Symposium on Compiler Construction*, 1984.