Generalizing Data Flow Information

August, 2007

skape
mmiller@hick.org

Abstract

Generalizing information is a common method of reducing the quantity of data that must be considered during analysis. This fact has been plainly illustrated in relation to static data flow analysis where previous research has described algorithms that can be used to generalize data flow information. These generalizations have helped support more optimal data flow analysis in certain situations. In the same vein, this paper describes a process that can be employed to generalize and persist data flow information along multiple *generalization tiers*. Each generalization tier is meant to describe the data flow behaviors of a conceptual software element such as an instruction, a basic block, a procedure, a data type, and so on. This process makes use of algorithms described in previous literature to support the generalization of data flow information. To illustrate the usefulness of the generalization process, this paper also presents an algorithm that can be used to determine reachability at each generalization tier. The algorithm determines reachability starting from the least specific generalization tier and uses the set of reachable paths found to progressively qualify data flow information for each successive generalization tier. This helps to constrain the amount of data flow information that must be considered to a minimal subset.

1 Introduction

Data flow analysis uses data flow information to solve a particular data flow problem such as determining reachability, dependence, and so on. The algorithms used to obtain data flow information may vary in terms of accuracy and precision. To help quantify effectiveness, data flow algorithms may generally be categorized based on specific sensitivities. The first category, referred to as *flow sensitivity* is used to convey whether or not an algorithm takes into account the implied order of instructions. *Path sensitivity* is used to convey whether or not an algorithm considers predicates. Finally, algorithms may also be *context-sensitive* if they take into account a calling context to restrict analysis to realizable paths when considering interprocedural data flow information.

Data flow information is typically collected by statically analyzing the data dependence of instructions or statements. For example, conventional def-use chains describe the variables that exist within $in()$, $out()$, $use()$, $def()$, and $kill()$ set for each instruction or statement. Understanding data flow information with this level of detail makes it possible to statically solve a particular data flow problem. However, the resources needed to represent the def-use data flow information can be prohibitive when working with large applications. Depending on the data flow problem, the amount of data flow information required to come to a solution may be in excess of the physical resources present on a computer performing the analysis. This physical resource problem can be solved using at least two general approaches.

The most basic approach might involve simply *partitioning*, or *fragmenting*, analysis information such that smaller subsets are considered individually rather than attempting to represent the complete set of data flow information at once[15]. While this would effectively constrain the amount of physical resources required, it would also directly impact the accuracy and precision of the underlying algorithm used to perform data flow analysis. For instance, identifying the "interesting portion" of a program may require more state than can be feasibly obtained in single program fragment. A second and potentially more optimal approach might involve *generalizing* data flow information. By generalizing data flow information, an algorithm can operate within the bounds of physical resources by making use of a more abstract view of the complete set of data flow information. The distinction between the generalizing approach and the partitioning approach is that the generalized data flow information should not affect the accuracy of the algorithm since it should still be able to represent the complete set of generalized data flow information at once.

There has been significant prior work that has illustrated the effectiveness of generalizing data flow information when performing data flow analysis. The def-use information obtained between instructions or statements has been generalized to describe sets for basic blocks. Horwitz, Reps, and Binkley describe how a *system dependence graph* (SDG) can be derived from intraprocedural data flow information to produce a summary graph which convey context-sensitive data flow information at the procedure level[7]. Their paper went on to describe an interprocedural slicing algorithm that made use of SDGs. Reps, Horwitz, and Sagiv later described a general framework (IFDS) in which many data flow analysis problems can be solved as graph reachability problems[13, 14]. The algorithms proposed in their paper focus on restricting analysis to interprocedurally realizable paths to improve precision. Identifying interprocedurally realizable paths has since been compared to the concept of context-free-language (CFL) reachability (CFL-reachability)[8]. These algorithms have helped to form the basis for techniques used in this paper to both generalize and analyze data flow information.

This paper approaches the generalization of data flow information by defining *generalization tiers* at which data flow information can be conveyed. A generalization tier is intended to show the data flow relationships between a set of conceptual software elements. Examples of software elements include an instruction, a basic block, a procedure, a data type, and so on. To define these relationships, data flow information is collected at the most specific generalization tier, such as the instruction tier, and then generalized to increasingly less-specific generalization tiers, such as the basic block, procedure, and data type tiers. Figure 1 provides a visual representation of some of the generalization tiers which may exist for data flow information collected from a program. The concentric rings are meant to illustrate containment.

To illustrate the usefulness of generalizing data flow information, this paper also presents a progressive algorithm that can be used to determine reachability between nodes on a data flow graph at each generalization tier. The algorithm starts by generating a data flow graph using data flow information from

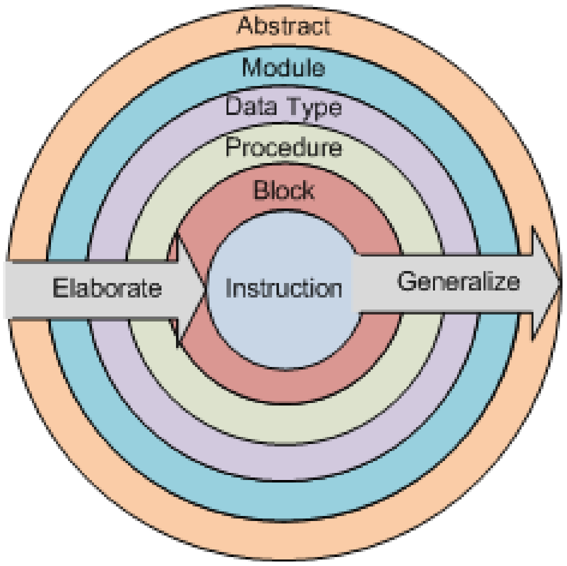

Figure 1: Logical generalization tiers at which data flow information can be described. Elaborate from least-specific to most-specific and generalize from most-specific to least-specific.

the least-specific generalization tier. The graph is then analyzed using a previously describe algorithm to determine reachability between an arbitrary set of nodes. The set of reachable paths found is then used to qualify the set of more-specific potentially reachable paths found at the next generalization tier. The more-specific paths are used to construct a new data flow graph. These steps then repeat using each more-specific generalization tier until it is not possible to obtained more detailed information. The benefit of this approach is that a minimal set of data flow information is considered as a result of progressively qualifying data flow paths at each generalization tier. It should be noted that different reachability problems may require state that is prohibitively large. As such, it is helpful to consider refining a reachability problem to operate more efficiently by making use of generalized information.

This paper is organized into two sections. §2 discusses the algorithms used to generalize data flow information at each generalization tier. §3 describes the algorithm used to determine reachable data flow paths by progressively analyzing data flow information at each generalization tier. It should be noted in advance that the author does not claim to be an expert in this field; rather, this paper is simply an explanation of the author's current thoughts. These thoughts attempt to take into account previous work whenever possible to the extent known by the author. Given that this is the case, the author is more than willing to receive criticism relating to the the ideas put forth in this paper.

2 Generalization

Generalizing data flow information can make it possible to analyze large data sets without losing accuracy. This section describes the process of generalizing information at each generalization tier. As a matter of course, each generalization tier uses data flow information obtained from its preceding more specific generalization tier. In this way, the basic block tier generalizes information obtained at the instruction tier, the procedure tier generalizes information obtained at the basic block tier, and so on. The algorithms used to generalize information at each generalization tier can have a direct impact on the accuracy of the information that can be obtained when used during data flow analysis. The subject of accuracy will be addressed for each specific tier.

To obtain generalized data flow information, a set of target executable image files, or *modules*, must be defined. The target modules serve to define the context from which data flow information will be obtained and generalized. The general process used to accomplish this involves visiting each procedure within each module. For each procedure, data flow information is collected at the instruction tier and is then generalized to each less-specific tier. To facilitate the reachability algorithm, it is assumed that as the data flow information is collected, it is persisted in a form such that can be accessed on demand. The process described in this paper assumes a normalized database is used to contain the data flow information found at each generalization tier. In this manner, the upper limit associated with the number of target modules is tied to the amount of available persistent storage with respect to the amount required by a given data flow problem.

Before proceeding, it is important to point out that while this paper describes explicit algorithms for generalizing at each tier, it is entirely possible to substitute alternative algorithms. This serves to illustrate that the concept of generalizing information along generalization tiers is sufficiently abstract enough to support representing alternate forms of data flow and control flow information. By using different algorithms, it is possible to convey different forms of data flow relationships which vary in terms of precision and accuracy.

2.1 Instruction Tier

Generalizing data flow information presupposes that there is data flow information to generalize. As such, a base set of data flow information must be collected first. For the purposes of this paper, the most specific data flow information is collected at the instruction tier using the *Static Single Assignment* (SSA) implementation provided by Microsoft's Phoenix framework, though other algorithms could just as well be used[11]. SSA is an elegant solution to the problem of representing data flow information in a flow-sensitive manner. Each definition and use of a given variable are defined in terms of a unique variable version which makes it possible to show clear, unambiguous data flow relationships. In cases where data flow information may merge along control flow paths, SSA makes use of a *phi* function which acts as a pseudo-instruction to represent the merge point. Obtaining distinct data flow paths at the instruction tier can be accomplished by traversing an SSA graph for a given procedure starting from each root variable, which have no prior definitions, and proceeding to each reachable leaf variable, which have no subsequent uses, are encountered along each data flow path. The end result of this traversal is the complete set of data flow paths found within the context of a given procedure.

One of SSA's limitations is that it is only designed to work intraprocedurally and therefore makes no effort to describe the behavior of passing data between procedures, such as through input and output parameters. In order to provide an accurate, distinct path data flow representation, one must take into account interprocedural data flow. One method of accomplishing this is to generalize the concept of SSA's *phi* function and use it represent formal parameters. In this way, the *phi* function can be used to represent data flow merges that happen as a result of data passing as input or output parameters when a procedure is called. A *phi* function can be created to represent each formal input and output parameter for a procedure, thus linking definitions of parameters at a call site to actual parameter uses in a callee. Reps, Horwitz, and Sagiv describe a concept similar to this[13].

In addition to using *phi* functions to link the definitions and uses of formal parameters, it is also necessary to *fracture* data flow paths at call sites that are found within a procedure . This is necessary because data flow paths collected using SSA information will convey a relationship between the input parameters passed to a procedure and the output parameters returned by a procedure. This is the case because a *call* instruction at a call site appears to use input parameters and define output parameters, thus creating an implicit link between input and output parameters. Since SSA information is obtained intraprocedurally, it is not possible to know in advance whether or not an input parameter will influence an output parameter.

To fracture a data flow path, the instructions that define input parameters passed at a given call site are instead linked directly to the associated formal input parameter *phi* functions that are found in the context of the target procedure. Likewise, instructions that use output parameters previously defined by the call instruction are instead linked directly to the associated formal output parameter *phi* functions found in the context of the target procedure. This has the effect of breaking the original data flow path into two disconnected data flow paths at the call site location.

The linking of actual parameters and call site parameters with formal parameters has been illustrated in previous literature. Horwitz, Reps, and Binkley used this concept during the construction of a *system dependence graph* (SDG)[7]. The concept of creating symbolic variables that are later used to link information together is not new[15]. Figure 2 provides an example of what a conventional and fractured data flow path might look like.

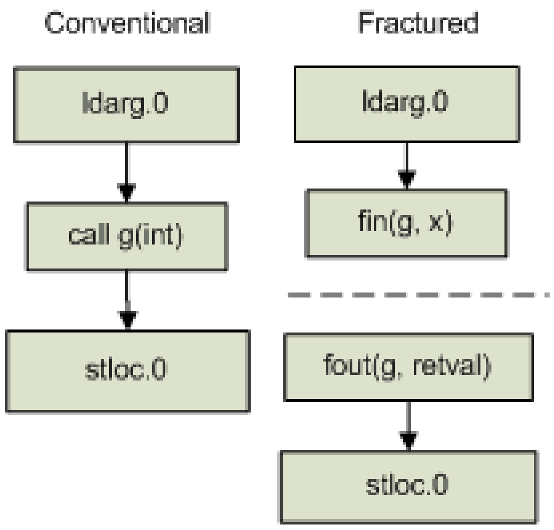

Figure 2: Fracturing a data flow path at a call site. Call instructions no longer act as the receivers or producers of data that is passed between procedures. Instead, formal parameters represent interprocedural *phi* functions.

Using the fracturing concept, the instruction tier's path-sensitive data flow information for a given procedure becomes disconnected. This helps to improve the overall accuracy of the data flow paths that are conveyed. Fracturing also has the added advantage of making it possible to use formal parameter *phi* functions to dynamically link a caller and a callee at runtime. This makes it possible to identify context-sensitive interprocedural data flow paths at the granularity of an instruction. This ability will be described in more detail when the reachability algorithm is described in §3.

With an understanding of the benefits of fracturing, it is now possible to define the general form that data flow paths may take at the instruction tier. This general form is meant to describe the structure of data flow paths at the instruction tier in terms of the potential set of origins, transient, and terminal points with respect to the general instruction types. Based on the description given above, it is possible to categorize instructions into a few general types. Using these general instruction types, the general form of instruction data flow paths can be captured as illustrated by the diagram in figure 3.

1. *value*: Defines or uses a data value
2. *compare*: Compares a data value
3. *fin*: Pseudo instruction representing a formal input parameter
4. *fout*: Pseudo instruction representing a formal output parameter

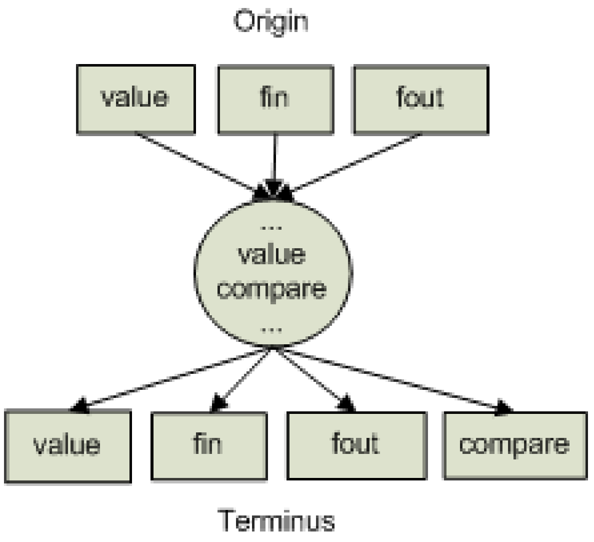

Figure 3: General forms of data flow paths at the instruction tier.

Based on this general description of instruction data flow paths, it is helpful to consider a concrete example. Consider the example source code described below which shows the implementation of the *f* function.

```
static public int f(int x)
{
   return (x > 0) ? g(x) : x + 1;
}
```

This function is intentionally very simple so as to limit the number of data flow paths that must be represented visually. Using the concepts described above, the instruction data flow paths that would be created as a result of analyzing this procedure are shown in figure 4. Note that the call site for the *g* function results in two disconnected data flow paths. The end result is that there are four unique data flow paths within this procedure, each denoted by a unique edge color.

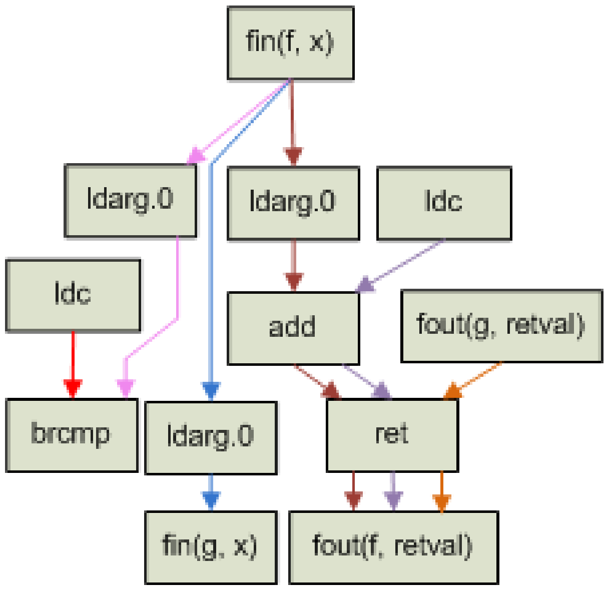

Figure 4: Instruction tier data flow paths for the example code. The context for these data flow paths is the *f* function.

2.2 Basic Block Tier

Once the complete set of data flow paths are identified at the instruction tier for a given procedure, the next step is to generalize data flow information to the basic block tier. At the basic block tier, instruction data flow paths should be generalized to show path-sensitive data flow interactions between basic blocks rather than instructions. This level of generalization reduces the amount of information needed to represent data flow paths. For example, there are many cases where data will be passed between multiple instructions within the same basic block. Using basic block tier generalization, those individual operations can be generalized and represented as a single basic block. The generalized basic block data flow paths can then be persisted for subsequent use when determining reachability in much the same fashion that was used at the instruction tier.

Since the instruction tier's data flow information has been fractured and parameters passed at call sites have been tied to *phi* functions, an approach must be defined to preserve this information at the basic block tier during generalization. An easy way of preserving this information is to define the formal parameters which represent input and output parameters as being contained within distinct pseudo blocks. For example, the *phi* functions representing formal input parameters can exist within a *formal entry* pseudo block. Likewise, the *phi* functions representing formal output parameters can exist within a *formal exit* pseudo block. Both pseudo blocks can then be tied to the procedure associated with the formal parameters. Defining the underlying instruction tier *phi* functions in this way makes it trivial to retain information that will be needed to define context-sensitive interprocedural data flow at less-specific generalization tiers. Like the instruction tier, it is possible to dynamically link data passed to a pseudo block in a caller's context to subsequent uses in a callee's context. Figure 5 shows the general form that basic block data flow paths may take.

The act of generalizing instruction data flow paths means that two or more distinct instruction data flow paths may produce the same basic block data flow path. When this occurs, only one basic block data flow path should be defined since it will effectively capture the information conveyed by the set of distinct instruction data flow paths. Each corresponding instruction data flow path should still be associated with a single basic block data flow path. This association makes it possible to show the set of instruction data flow paths that have been generalized by a specific basic block data flow path. The association can be persisted in a normalized database by creating a one-to-many link table between basic block and instruction data flow paths. Figure 6 provides an example of what would happen when generalizing the instruction data flow paths described in figure 4.

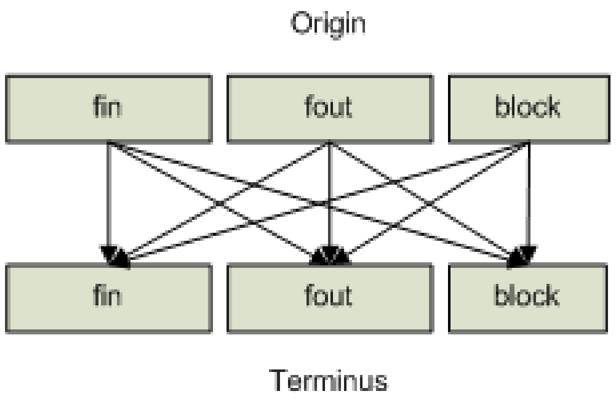

Figure 5: General forms of data flow paths at the basic block tier.

2.3 Procedure Tier

Generalizing data flow paths from the basic block tier to the procedure tier further reduces the amount of information needed to show data flow behavior. Procedure tier data flow paths are meant to show how data is passed between procedures through formal parameters. This covers scenarios such as passing a procedure's formal input parameter to a child procedure's formal input parameter, using the formal output parameter of a child procedure as the formal in-

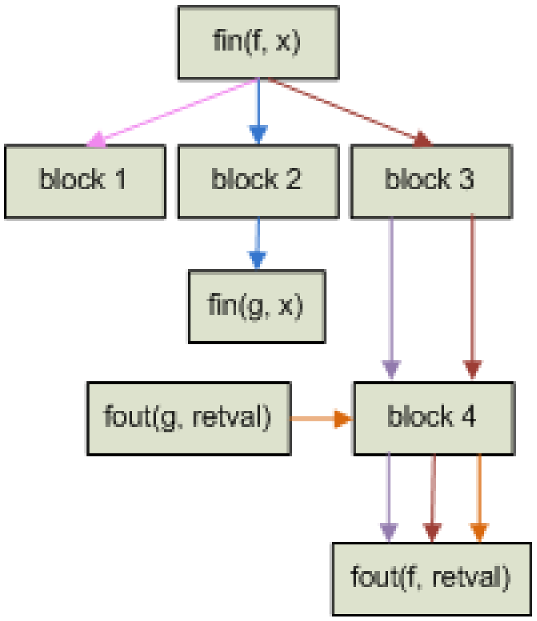

Figure 6: Basic block tier data flow paths obtained by generalizing the instruction data flow paths described in figure 4. The context for these data flow paths is the f function.

put parameter to another called procedure, and so on. These behaviors are all represented within the context of a particular procedure.

Based on these constraints, only two classes ofbasic block data flow paths need to be considered. The first class involves data traveling from any block to a formal input or output parameter, thus showing interprocedural flows. The second class involves data traveling from a formal input or formal output parameter to a terminal point in a procedure. This effectively eliminates any intraprocedural data flows that are not carried over to another procedure in some form. Since data flow information about which formal parameters are used or defined is conveyed by basic block data flow paths, it is possible to simply generalize this data flow information to show data flowing to formal parameters within the context of a given procedure. While it may be tempting to think that one must only show data flow paths between two formal parameters, it is also necessary to show data flow paths that originate from data that is locally defined within a procedure, such as through a local variable which is not populated by a formal parameter. As such, the general form that data flow paths may take at the procedure tier is illustrated by figure 7. Figure 8 provides an example of what would happen when generalizing the basic block data flow paths described in figure 6.

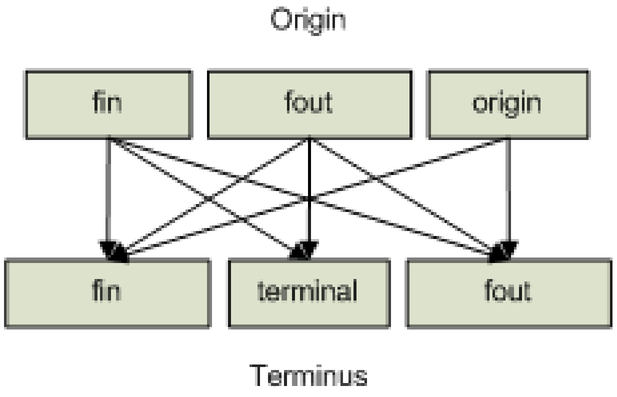

Figure 7: General forms of data flow paths at the procedure tier.

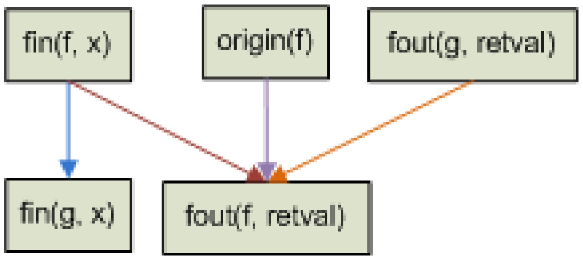

Figure 8: Procedure tier data flow paths obtained by generalizing the basic block data flow paths described in figure 4. The context for these data flow paths is the f function.

Procedure data flow paths may generalize multiple basic block data flow paths and thus can make use of a one-to-many link table to illustrate this association. While generalizing data flow paths to the procedure tier is trivial, the challenging aspect comes when determining reachability. This will be discussed in more detail in §3.

2.4 Data Type Tier

Using data flow information obtained from the procedure tier, it is sometimes possible, depending on language features, to generalize data flow information to the data type tier. Generalizing to the data type tier is meant to show how formal parameters are passed between data types within the context of a given data type. This relies on the underlying language having the ability to associate procedures with data types. For example, object-oriented languages are all capable of associating procedures with data types, such as through classes defined in C++, C#, and other languages. In the case of languages where data types do not have procedures, it may instead be possible to associate procedures with the name of the source file that contains them. In both cases, it is possible to show formal parameters passing between elements that act as containers for procedures, regardless of whether the underlying elements are true data types.

The benefit of generalizing data flow information at the data type tier is that it helps to further reduce the amount of data flow information that must be represented. Since the small example source code that has been used to illustrate generalizations at each tier only involves passing formal parameters within the same data type, it is useful to consider an alternative example which involves passing data between multiple data types.

```
class Company {
   void AddEmployee(int num) {
      Person employee = new Person(num);
      employees.Add(employee);
      Console.WriteLine("New employee {0}", employee);
   }
   int EmployeeCount() {
      return employees.Count;
   }
   private ArrayList employees;
}
```

Figure 9 shows the data type data flow paths for the example code shown above. It is important to note that unlike previous tiers, the specific formal parameters that are being passed between types is not preserved. Instead, only the fact that formal parameters are passed between data types is retained. In this manner, *fin* indicates a data type's formal input parameter and *fout* indicates a data type's formal output parameter.

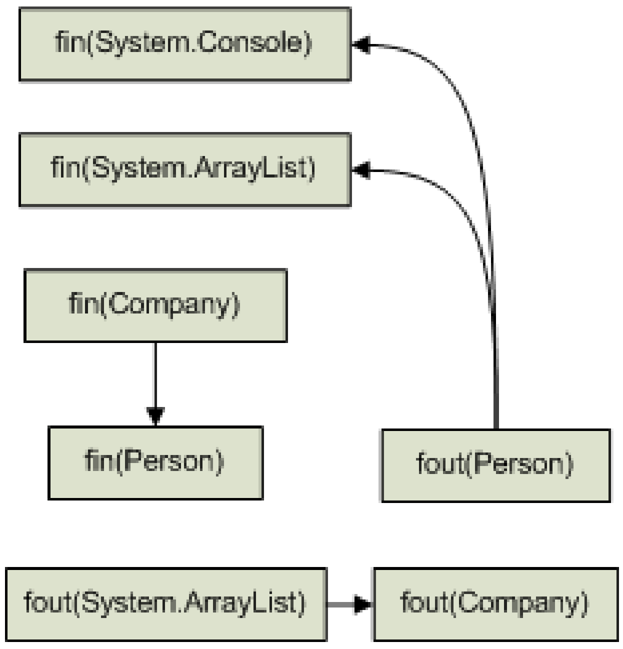

Figure 9: Data type tier data flow paths obtained by generalizing the procedure tier data flow paths. The context for these data flow paths is the *Company* data type.

In a fashion much the same as previous generalization tiers, a single data type data flow path can represent multiple underlying procedure data flow paths. Each generalized procedure data flow path can be associated with its corresponding data type data flow path through a one-to-many link table in a normalized database.

2.5 Module Tier

Generalizing data flow information to the module tier is meant to show how data flows between distinct modules. As with each step in the generalization process, the module tier data flow paths lose much of the information that is conveyed at more specific tiers. Figure 10 shows module tier data flow paths that would be defined when generalizing the data type data flow paths illustrated in figure 9.

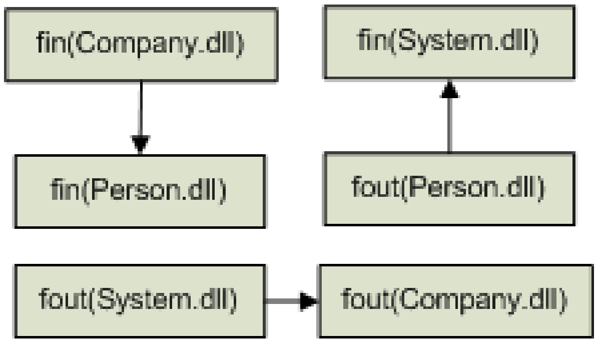

Figure 10: Module tier data flow paths obtained by generalizing the data type tier data flow paths. The context for these data flow paths is the *Company.dll* module.

2.6 Abstract Tiers

Once data flow paths have been generalized from the instruction tier through the module tier, it is no longer possible to create additional concrete generalizations for most runtime environments[1]. Even though it may not be possible to establish concrete generalizations, it is possible to define abstract generalizations. An abstract generalization attempts to show data flow relationships between abstract elements. A good example of an abstract element would be a logical *component* which is defined in the architecture of a given application. For example, a VPN client application might be composed of a user interface component and a networking component, each of which may consist of multiple concrete modules. By defining logical components and associating concrete modules with each component, it is possible to further generalize information beyond the module tier.

[1]An exception to this is managed code which has an additional concrete *assembly* tier

Given the example described above, it may be prudent to define two abstract generalization tiers. The first abstract tier is the *component* tier. In this context, a component is defined as a logical software component that contains one or more concrete modules. The component tier makes it possible to illustrate data flow between conceptual components within an application as derived from how data flows between concrete modules. The second abstract tier is the *application* tier. The application tier can be used to illustrate how data is passed between conceptual applications. For example, a web browser application passes data in some form to a web server application, both of which consist of conceptual components which, in turn, consist of concrete modules.

The caveat with abstract generalization tiers is that it must be possible to illustrate data flow between what may otherwise be disjoint concrete elements. The reason for this is that, often times, the paths that data will take between two modules which belong to different logical components will be entirely indirect with respect to one another. For this reason, it is necessary to devise a mechanism to *bridge* data flow paths between concrete software elements that belong to each logical component or application. A particularly useful example of an approach that can be taken to bridge two distinct components can be found in web services.

In a web services application, it is often common to have a client component and a server component. The two components pass data to one another through an indirect channel, such as through a web request. For this reason, it is not immediately possible to show direct data flow paths from a web client component to a web service component. To solve this problem, one can define a mechanism that bridges the formal parameters associated with the web service method that is being invoked. In this manner, the the formal input parameters for a web service method found on the client side can be implicitly linked and shown to define the formal input parameters received on the web service side. By illustrating data flow at a concrete tier, it is possible to generalize data flow behaviors all the way up through the abstract tiers.

The benefit of describing data flow behavior at abstract tiers is that it makes it possible to derive data flow behaviors between abstract software elements rather than strictly focusing on concrete software elements. This is useful when attempting to view an application's behavior at a glance rather than worrying about the specific details relating to how data is passed. For example, this could be used to help validate threat models which describe how data is expected to be passed between abstract components within an application.

When generalizing information at abstract tiers, the only information that can be conveyed, at least based on the approach described thus far, is whether or not a component or application are passing data through a formal input or formal output parameter. The specifics of which formal parameters are passed is no longer available for use in generalization. Using the example shown at the data type tier, one might assume the following component associations: *Company.dll* and *Person.dll*, which contain the `Company` data type and `Person` data type, are part of the user interface component of a human resources application. The classes used from system libraries can be generically grouped as belonging to an *external library* component. Using these, groupings, the component data flow paths may be represented as shown in figure 11.

As with all previously described generalization tiers, a single component data flow path may represent multiple module data flow paths. The single component data flow path should be associated with each corresponding module data flow path through a one-to-many link table in a normalized database.

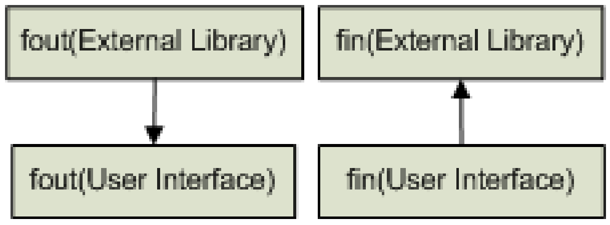

Figure 11: Component tier data flow paths obtained by generalizing the module tier data flow paths. The context for these data flow paths is the *user interface* component.

3 Reachability

The real benefit of the generalizations described in §2 can be realized when attempting to solve a graph reachability problem. By generalizing data flow behaviors to both abstract and concrete generalization tiers, it is possible to reduce the amount of information that must be represented when attempting to determine graph reachability. This is further improved by the fact that data flow paths found at less-specific generalization tiers can be used to progressively qualify potential data flow paths at more-specific generalization tiers. This qualification is possible due to the fact that less-specific data flow paths are associated with more-specific data flow paths at each generalization tier through a one-to-many link table, thus permitting trivial expansion. The benefit of qualifying data flow paths in this fashion is that only the minimal set of information needed to determine reachability must be considered at once at each generalization tier. This can drastically reduce the physical resources required to solve a graph reachability problem by effectively limiting the size of a graph. This general approach is captured by the *Progressive Qualified Elaboration* (PQE) algorithm described by 3.1. This concept is very similar to the ideas outlined by Schultes' *highway hierarchy* which is used to optimize fast path discovery when identifying travel routes in road networks[16].

For the purposes of this paper, graph reachability is restricted to determining realizable paths between

Algorithm 3.1: PQE$(E, D_{source}, D_{sink})$

```
$P \leftarrow \emptyset$
$G \leftarrow BuildGraph(E)$
while $G \neq \emptyset$
    do
        $V_{source} \leftarrow Vertices(G, D_{source})$
        $V_{sink} \leftarrow Vertices(G, D_{sink})$
        $P \leftarrow Reachability(G, V_{source}, V_{sink})$
        $P_{elaborated} \leftarrow Elaborate(P)$
        $G \leftarrow BuildGraph(P_{elaborated})$
return $(P)$
```

two *flow descriptors*: a *source* and a *sink*. A flow descriptor provides information that is needed to identify corresponding vertices within a graph at each generalization tier. The tables in figure 12 and figure 13 show the information needed to identify source and sink vertices at each generalization tier for the example that will be described in this section.

The `PQE` algorithm itself requires three parameters. The first parameter, E, contains the set of generalized elements to be analyzed. For example, it may contain the set of target modules that should be analyzed. The second and third parameters, D_{source} and D_{sink}, represent the source and sink flow descriptors, respectively.

The first step taken by the algorithm is to define P as an empty set. P will be used to contain the set of reachable paths between an actual set of sources and sinks at a given generalization tier. After P has been initialized, G is initialized to a flow graph that conveys data flow relationships between the set of elements provided in E. The approach taken to construct the flow graph involves retrieving persisted data flow information for the appropriate generalization tier. Once P and G have been initialized, the qualified elaboration process can begin.

For each loop iteration, a check is made to see if G is an empty graph (contains no vertices). If G is empty, the loop terminates. If G is not an empty graph, reachable paths between the actual set of sources and sinks are determined. This is accomplished by first identifying the vertices in G that are associated with the flow descriptors D_{source} and D_{sink} at the current generalization tier. The actual set of sources and sinks found to be associated with these descriptors are stored in V_{source} and V_{sink}, respectively. With the set of actual source and sink vertices identified, a reachability algorithm, *Reachability*(), can be used to determine the set of reachable paths in flow graph G between the two sets of vertices, V_{source} and V_{sink}. The result of this determination is stored in P. The final step in the iteration involves using *qualified elaboration* to construct a new flow graph containing more-specific data flow paths which are qualified by the set of data flow paths encountered in the reachable paths found in P. This set is then elaborated to a subset that contains the associated data flow paths from the next, more specific tier, such as by elaborating to a subset of basic blocks data flow paths from a more general set of procedure data flow paths. The result of the elaboration is stored in $P_{elaborated}$. Finally, a new flow graph is constructed and stored in G using the elaborated set of flow paths contained within $P_{elaborated}$.

When it is not possible to obtain a more-detailed flow set, such as when the instruction tier is reached, an empty graph is created and the algorithm completes by returning P. In the final iteration, P contains the most detailed set of reachable data flow paths found between the source and sink flow descriptors. The benefit of approaching graph reachability problems in this fashion is that only a subset of the elements at any generalization tier need to be considered at once. These subsets are dictated by the set of reachable data flow paths found at each preceding generalization tier. In this manner, the subset of procedure data flow paths that need to be considered would be effectively qualified by the set of data types and modules found to be involved in data flow paths between the source and sink flow descriptors at less-specific tiers.

For the purposes of this paper, the *Reachability*() algorithm is designed to consider realizable paths at each generalization tier in manner that is similar to the concept described by Reps et al[13]. This involves traversing the graph in context-sensitive fashion. To accomplish this, the algorithm keeps a scope stack at each generalization tier. The scope may be an assembly, a type, or a procedure. When data is passed through to a formal input parameter, the scope for the formal input parameter is pushed onto the stack. When data is returned through a formal output parameter to another location, the algorithm ensures that the scope that is being returned to is the parent scope. In this way, only realizable paths are considered at each generalization tier which limits the number of paths that must be considered and also has the benefit of producing more accurate results.

The specific algorithm used for the *Elaborate*() function involves using the set of data flow paths found at a less-specific tier to identify the set of more-specific data flow paths that have been generalized. This is accomplished by simply using the one-to-many link tables that were populated during generalization to determine the subset of data flow paths that must be considered at the next generalization tier. For example, elaborating from a set of procedure data flow paths would involve determining the complete set of basic block data flow paths that have been generalized by the affected set of procedure data flow paths.

Based on this general description of the algorithm, it is useful to consider a concrete example This section provides a concrete illustration by determining reachability between a source and a sink using an example web application that consists of a web client and a web service component. This is illustrated by progressively drilling down through each generalization tier starting from the least-specific tier, the abstract tier, and working toward the most-specific tier, the instruction tier. At each tier, a description of the number of data flow paths that must be represented and the number of reachable data flow paths found is given. This particular example will attempt to determine concrete reachable data flow paths between the return value of `HttpRequest.get_QueryString` and the first formal input parameter of `Process.Start`. The implications of a reachable path between these two points could be indicative of a command injection vulnerability within the application. The tables in figure 12 and figure 13 show the flow descriptors for the source and sink, respectively. These flow descriptors are used to identify associated vertices at each generalization tier.

Tier	**Information**
Component	`fout(Undefined)`
Module	`fout(System.Web.dll)`
Data Type	`fout(System.Web.HttpRequest)`
Procedure	`fout(get_QueryString, 0)`
Basic Block	`fout(get_QueryString, 0)`
Instruction	`fout(get_QueryString, 0)`

Figure 12: Source flow descriptor for the return value of `HttpRequest.get_QueryString`

Tier	**Information**
Component	`fin(Undefined)`
Module	`fin(System.dll)`
Data Type	`fin(System.Diagnostics.Process)`
Procedure	`fin(Process.Start, 0)`
Basic Block	`fin(Process.Start, 0)`
Instruction	`fin(Process.Start, 0)`

Figure 13: Sink flow descriptor for the first (zero-indexed) formal input parameter of `Process.Start`

For this illustration, there is in fact a data flow path that exists from the source descriptor to the sink descriptor. However, unlike conventional data flow paths, this data flow path happens to cross an abstract boundary between the two components. In this case, data is passed from the web client component through an HTTP request to a method hosted by the web service component. This path can be seen by first looking at a portion of the web client code:

```
class Program {
   static void Main(string[] args) {
      HttpRequest request = new HttpRequest(
         "a","b","c");
      WebClient client = new WebClient();

      client.ExecuteCommand(
         request.QueryString["abc"]);
   }
}
```

```
[WebServiceBinding]
public class WebClient : SoapHttpClientProtocol {
   [SoapDocumentMethod]
   public void ExecuteCommand(string command) {
      Invoke("ExecuteCommand",
         new object[] { command });
   }
}
```

In this contrived example, data is shown as being passed from a query string obtained from what is presumably a real HTTP request to the client portion of the web service method *ExecuteCommand*. The web service application, in turn, contains the following code:

```
[WebService]
public class WebService {
   [WebMethod]
   public void ExecuteCommand(string command) {
      System.Diagnostics.Process.Start(command);
   }
}
```

In conventional tools, it would not be possible to directly model this data flow path because the data flow path is indirect. However, using a simple methodology to bridge the client-side formal input parameters with the server-side formal input parameters at the instruction tier, it is possible to connect the two and represent data flow between the two conceptual software elements at each generalization tier. The following sections will provide visual examples of how the PQE algorithm narrows down and eliminates unnecessary data flow paths at each generalization tier by progressively qualifying data flow information. One thing to note about the graphs at each tier is that implicit edges have been created between formal input and output parameters that reside in external (un-analyzed) libraries. This is done under the assumption that a formal input parameter may affect a formal output parameter in some way in the context of the code that is not analyzed. If all target code paths have been analyzed, then this is not necessary. The graphs shown at each tier were automatically generated but have been modified to allow them to fit within the margins of this document and in some cases highlight important features.

3.1 Abstract Tiers

Abstract tiers represent the most general view of the data flow behaviors of an application. The data flow behavior is modeled with respect to abstract software elements, such as a component, rather than concrete software elements, such as a module or a type. For this example, it is assumed that the PQE algorithm begins by modeling data flow behaviors between conceptual components in a web application. The web application is composed of two manually defined abstract components, a *Web Client* and a *Web Service*. These two components both rely on external libraries, as represented by the *Undefined* component, which are outside of the scope of the application itself. When starting at the abstract tier, all abstract data flow paths must be considered as potential data flow paths. The component tier data flow graph for this application is shown in figure 14.

Using the data flow graph shown in figure 14, PQE uses the *Reachability*() algorithm to determine data flow paths between a formal output parameter in the *Undefined* component and a formal input parameter in the *Undefined* component. At this generalization tier, there are many different paths that can be taken between these two components. This effectively results in the qualification of nearly all of the assembly tier data flow paths. These data flow paths are used to represent the data flow graph at the assembly tier.

In this example, PQE offers no improvements at abstract tiers because it is a requirement that all abstract data flow information be represented. Since the amount of information required to represent abstract data flow is minimal, this is not seen as a deficiency. Furthermore, for this particular example, nearly all component data flow paths are found to be involved in reachable paths. At worst, this is indicative that for small applications, it may not be necessary to start the algorithm by looking at abstract data flow information. Instead, one might immediately progress to the module or data type tiers.

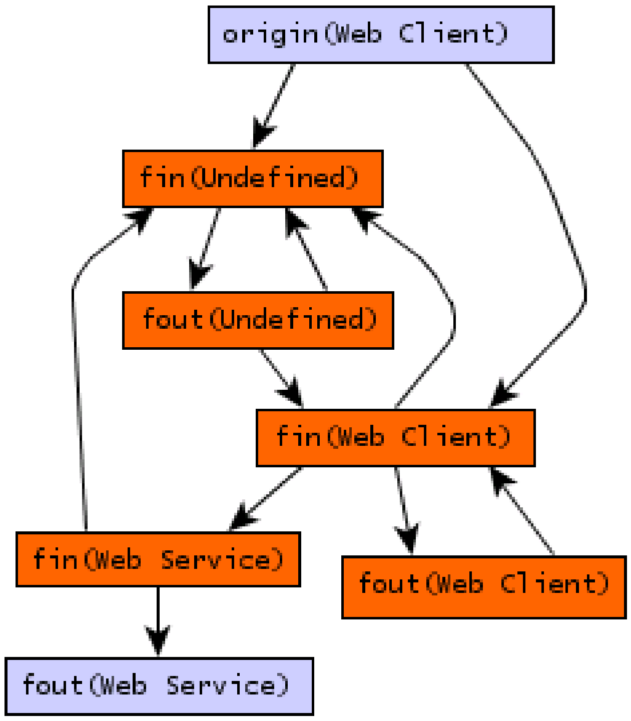

Figure 14: Complete component tier data flow graph for the web application. Component nodes involved in flow paths between the source and the sink have been manually annotated in dark orange.

3.2 Module Tier

The module tier uses the set of data flow paths found at the abstract component tier to construct a data flow graph that shows the data flow relationships between formal input and formal output parameters passed between modules. The graph is generated using the one-to-many table that was populated during generalization which conveys the module data flow paths that were generalized by the set of qualified component data flow paths. For this particular example, nearly all of the module data flow paths were qualified as potentially being involved in a reachable path between the source and sink flow descriptor. The graph that is generated as a result is shown in figure 15.

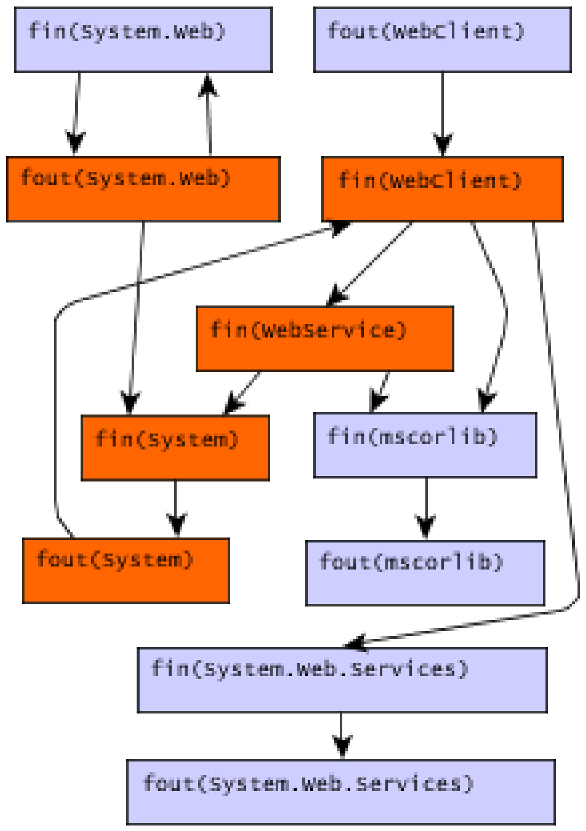

Figure 15: Module tier data flow graph for the web application representing potentially reachable paths between the source and sink flow descriptor. These paths were qualified by reachable paths found at the component tier. Module nodes involved in flow paths between the source and the sink have been manually annotated in dark orange.

Using this graph, the *Reachability*() algorithm is again employed to find paths between the source and sink flow descriptor at the module tier. In this case, only the edges between the nodes highlighted in dark orange are found to be involved in reachable paths between `fout(System.Web)` and `fin(System)`. The important thing to note is that even at the module tier, a data flow path is illustrated between `fin(WebClient)` and `fin(WebService)`. This will be a trend that will continue to each more specific generalization tier.

3.3 Data Type Tier

The data type tier uses the set of data flow paths found at the module tier to construct a data flow graph that shows the data flow relationships between formal input and formal output parameters passed between data types. The graph is generated using the one-to-many table that was populated during generalization which conveys the data type data flow paths that were generalized by the set of qualified module data flow paths. The graph that is generated as a result is shown in figure 16.

Using the graph, the *Reachability*() algorithm is again employed to find paths between the source and sink flow descriptor at the data type tier. Due to the simplicity of the example application, only a few data flow paths were rendered. The complete data flow path from `fout(System.Web.HttpRequest)` to `fin(System.Diagnostics.Process.Start)` can be clearly seen.

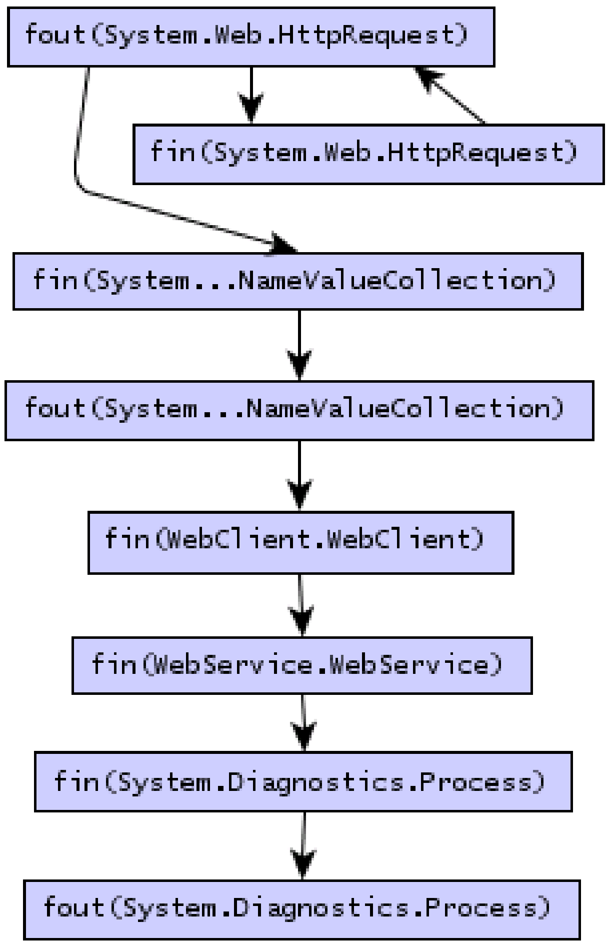
fout(System.Web.HttpRequest)
fin(System.Web.HttpRequest)
fin(System...NameValueCollection)
fout(System...NameValueCollection)
fin(WebClient.WebClient)
fin(WebService.WebService)
fin(System.Diagnostics.Process)
fout(System.Diagnostics.Process)

Figure 16: Data type tier data flow graph for the web application representing potentially reachable paths between the source and sink flow descriptor. These paths were qualified by reachable paths found at the module tier.

3.4 Procedure Tier

The procedure tier uses the set of data flow paths found at the data type tier to construct a data flow graph that shows the data flow relationships between formal input and formal output parameters passed between procedures. Unlike previous tiers, procedure tier data flow paths explicitly identify the formal parameter index that data is being passed to. This helps to further isolate data flow paths from one another and improves the overall accuracy of paths that are selected. The graph is generated using the one-to-many table that was populated during generalization which conveys the procedure data flow paths that were generalized by the set of qualified data type data flow paths. The graph that is generated as a result is shown in figure 17.

Using the graph, the *Reachability*() algorithm is again employed to find paths between the source and sink flow descriptor at the procedure tier. Due to the simplicity of the example application, only a few data flow paths were rendered. The complete data flow path from `fout(get_QueryString, 0)` to `fin(Start, 0)` can be clearly seen.

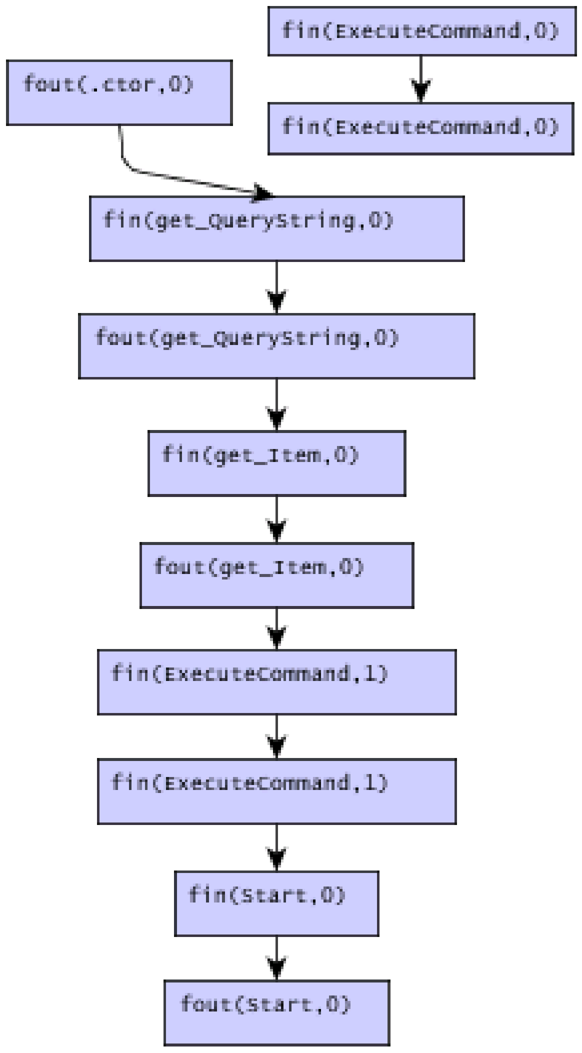

Figure 17: Procedure tier data flow graph for the web application representing potentially reachable paths between the source and sink flow descriptor. These paths were qualified by reachable paths found at the data type tier.

3.5 Basic Block Tier

The basic block tier uses the set of data flow paths found at the procedure tier to construct a data flow graph that shows the data flow relationships between formal input and formal output parameters passed between basic blocks. Like the procedure tier, basic block tier data flow paths also explicitly identify the formal parameter index that data is being passed to. The graph is generated using the one-to-many table that was populated during generalization which conveys the basic block data flow paths that were generalized by the set of qualified procedure data flow paths. The graph that is generated as a result is shown in figure 18. Due to the way that Phoenix currently represents basic blocks, the basic block tier data flow paths offer very little generalization beyond the instruction tier.

Using the graph, the *Reachability*() algorithm is again employed to find paths between the source and sink flow descriptor at the basic block tier. Due to the simplicity of the example application, only a few data flow paths were rendered. The complete data flow path from `fout(get_QueryString, 0)` to `fin(Start, 0)` can be clearly seen.

3.6 Instruction Tier

The instruction tier uses the set of data flow paths found at the basic block tier to construct a data flow graph that shows the data flow relationships between formal input and formal output parameters passed between instructions. Like the basic block tier, instruction tier data flow paths also explicitly identify the formal parameter index that data is being passed to. The graph is generated using the one-to-many table that was populated during generalization which conveys the instruction data flow paths that were generalized by the set of qualified basic block data flow paths. The graph that is generated as a result is shown in figure 19. The instruction tier data flow paths represent the final step taken by the algorithm as they contain the most specific description of data

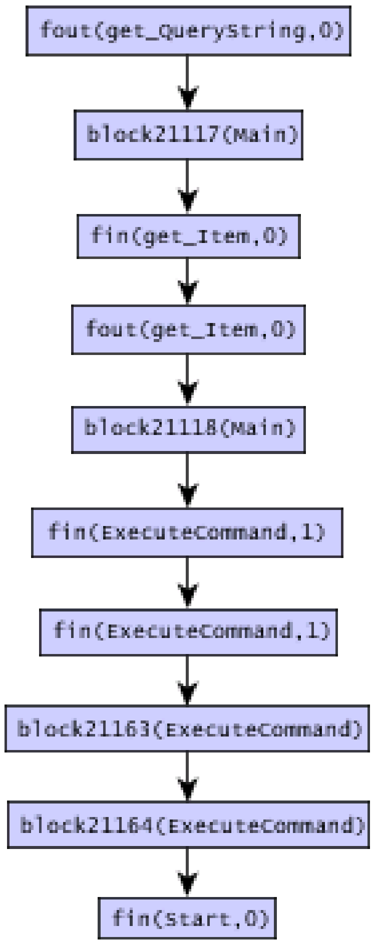

Figure 18: Basic block tier data flow graph for the web application representing potentially reachable paths between the source and sink flow descriptor. These paths were qualified by reachable paths found at the procedure tier.

flow paths.

Using the graph, the *Reachability*() algorithm is again employed to find paths between the source and sink flow descriptor at the instruction tier. Due to the simplicity of the example application, only a few data flow paths were rendered. The complete

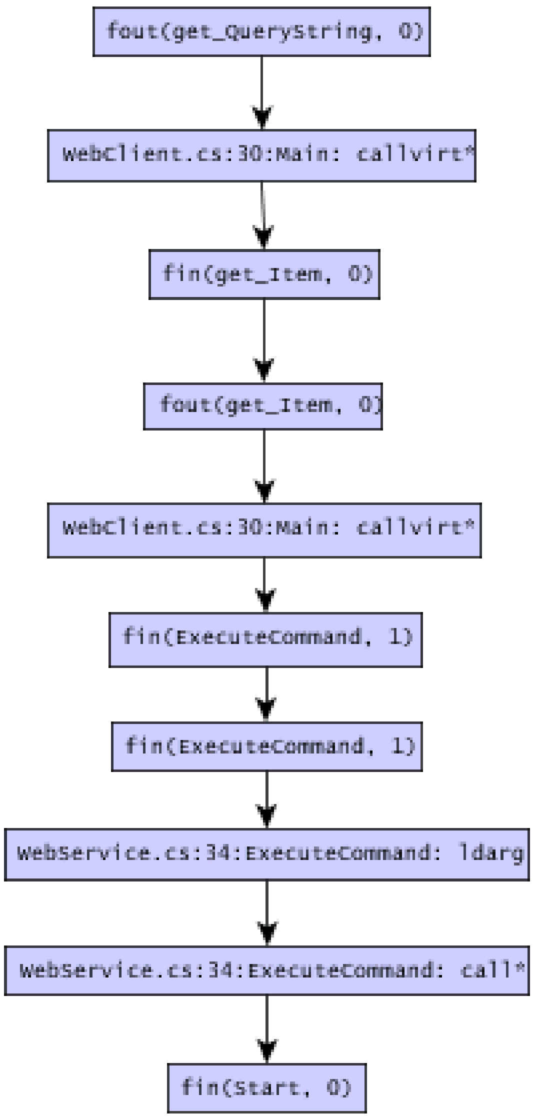
fout(get_QueryString, 0)
WebClient.cs:30:Main: callvirt*
fin(get_Item, 0)
fout(get_Item, 0)
WebClient.cs:30:Main: callvirt*
fin(ExecuteCommand, 1)
fin(ExecuteCommand, 1)
WebService.cs:34:ExecuteCommand: ldarg
WebService.cs:34:ExecuteCommand: call*
fin(Start, 0)

Figure 19: Instruction tier data flow graph for the web application representing potentially reachable paths between the source and sink flow descriptor. These paths were qualified by reachable paths found at the basic block tier.

data flow path from `fout(get_QueryString, 0)` to `fin(Start, 0)` can be clearly seen along with source lines that are encountered along the way.

4 Acknowledgements

The author would like to thank Rolf Rolles, Richard Johnson, Halvar Flake, Jordan Hind, and many others for thoughtful discussions and feedback.

5 Conclusion

This document has attempted to convey the potential benefits of generalizing data flow information along *generalization tiers*. Each generalization tier is used to represent the data flow behaviors of an abstract or concrete software element such as an instruction, basic block, procedure, and so on. Using this concept, data flow information can be collected at the most specific tier, the instruction tier, and then generalized to increasingly less-specific tiers. The generalization process has the effect of reducing the amount of data that must be considered at once while still conveying a general description of the manner in which data flows within an application.

Generalized data flow information can be immediately used in conjunction with existing graph reachability problems. For instance, a common task that involves determining reachable data flow paths between a conceptual *source* and *sink* location within an application can potentially benefit from operating on generalized data flow information. This paper has illustrated these potential benefits by defining the *Progressive Qualified Elaboration* (PQE) algorithm which can be used to progressively determine reachability at each generalization tier. By starting at the least specific generalization tier and progressing toward the most specific, it is possible to restrict the amount of data flow information that must be considered at once to a minimal set. This is accomplished by using reachable paths found at each generalization tier to qualify the set of data flow paths that must be considered at more specific generalization tiers.

While these benefits are thought to be present, the author has yet to conclusively prove this to be the case. The results presented in this paper do not prove the presumed usefulness of generalizing data flow information beyond the procedure tier. The author believes that analysis of large applications involving hundreds of modules could benefit from generalizing data flow information to the data type, module, and more abstract tiers. However, at the time of this writing, conclusive data has not been collected to prove this usefulness. The author hopes to collect information that either confirms or refutes this point during future research.

At present, the underlying implementation used to generate the results described in this paper has a number of known limitations. The first limitation is that it does not currently take into account formal parameters that are not passed at a call site, such as fields, global variables, and so on. This significantly restricts the accuracy of the data flow model that it is currently capable of generating. This limitation represents a more general problem of needing to better refine the underlying completeness of the data flow information that is captured.

While the algorithms presented in this paper were portrayed in the context of data flow analysis, it is entirely possible to apply them to other fields as well, such as control flow analysis. The PQE algorithm itself is conceptually generic in that it simply describes a process that can be employed to qualify the next set of analysis information that must be considered from a more generic set of analysis information. This may facilitate future research directions.

References

[1] Atkinson, Griswold. *Implementation Techniques for Efficient Data-flow Analysis of Large Programs.* Proceedings of the IEEE International Conference on Software Maintenance

(ICSM'01). 2001. http://www.cse.scu.edu/~atkinson/papers/icsm-01.ps

[2] Das, M. *Static Analysis of Large Programs: Some Experiences* 2000. http://research.microsoft.com/manuvir/Talks/pepm00.ppt

[3] Das, M., Lerner, S., Seigle, M. *ESP: Path-Sensitive Program Verification in Polynomial Time*. Proceedings of the SIGPLAN 2002 Conference on programming language design. 2002. http://www.cs.cornell.edu/courses/cs711/2005fa/papers/dls-pldi02.pdf

[4] Das, M., Fahndrich, M., Rehof, J. *From Polymorphic Subtyping to CFL Reachability: Context-Sensitive Flow Analysis Using Instantiation Constraints*. 2000. http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-99-84

[5] Dinakar Dhurjati1, Manuvir Das, and Yue Yang. *Path-Sensitive Dataflow Analysis with Iterative Refinement*. SAS'06: The 13th International Static Analysis Symposium, Seoul, August 2006.

[6] Erikson, Manocha. *Simplification Culling of Static and Dynamic Scene Graphs*. TR9809-009 by University of North Carolina at Chapel Hill. 1998. citeseer.ist.psu.edu/erikson98simplification.html

[7] Horwitz, S., Reps, T., and Binkley, D., Interprocedural slicing using dependence graphs. In Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation, (Atlanta, GA, June 22-24, 1988), ACM SIGPLAN Notices 23, 7 (July 1988), pp. 35-46.

[8] Horwitz, S., Reps, T., and Binkley, D., Retrospective: Interprocedural slicing using dependence graphs. 20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection, K.S. McKinley, ed., ACM SIGPLAN Notices 39, 4 (April 2004), 229-231.

[9] Gregor, D., Schupp, S. *Retaining Path-Sensitive Relations across Control Flow Merges*. Technical report 03-15, Rensselaer Polytechnic Institute, November 2003. http://www.cs.rpi.edu/research/ps/03-15.ps

[10] Kiss, Jasz, Lehotai, Gyimothy. *Interprocedural Static Slicing of Binary Executables*. http://www.inf.u-szeged.hu/~akiss/pub/kiss_interprocedural.pdf

[11] Microsoft Corporation. *Phoenix Framework*. http://research.microsoft.com/phoenix/

[12] Naumovich, G., Avrunin, G. S., and Clarke, L. A. 1999. Data flow analysis for checking properties of concurrent Java programs. In Proceedings of the 21st international Conference on Software Engineering (Los Angeles, California, United States, May 16 - 22, 1999). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 399-410.

[13] Reps, T., Horwitz, S., and Sagiv, M., Precise interprocedural dataflow analysis via graph reachability. In Conference Record of the 22nd ACM Symposium on Principles of Programming Languages, (San Francisco, CA, Jan. 23-25, 1995), pp. 49-61.

[14] Reps, T., Sagiv, M., and Horwitz S., Interprocedural dataflow analysis via graph reachability. TR 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark, April 1994.

[15] A. Rountev, B. G. Ryder, and W. Landi. Dataflow analysis of program fragments. In Proc. Symp. Foundations of Software Engineering, LNCS 1687, pages 235–252, 1999. http://citeseer.ist.psu.edu/rountev99dataflow.html

[16] Schultes, Dominik. *Fast and Exact Shortest Path Queries Using Highway Hierachies*. 2005. http://algo2.iti.uka.de/schultes/hwy/hwyHierarchies.pdf