Component-Level Dataflow Analysis

Atanas Rountev

Ohio State University
rountev@cse.ohio-state.edu

Abstract. Interprocedural dataflow analysis has a wide range of uses in software maintenance, testing, verification, and optimization. Despite the large body of research on various analyses, the widespread adoption of these techniques faces serious challenges. In particular, when software is built with reusable components, the standard approaches for dataflow analysis cannot be applied. This paper proposes a model of *component-level* analysis which generalizes the traditional model of *whole-program* analysis. We outline the theoretical foundations of component-level analysis, discuss some of the key technical challenges for such analysis, and present initial results from our work on addressing these challenges.

1 Introduction

Interprocedural dataflow analysis is a form of static program analysis that has been investigated widely in the last two decades. For example, many analyses have been developed for use in tools for *software understanding and maintenance*. Other areas in which dataflow analysis is commonly used are *software testing* and *software verification*, both of which are essential for producing high-quality systems. Last but not least, dataflow analysis continues to play an important role in the area of *performance optimization*, by enabling compiler optimizations for numerous programming languages and hardware architectures.

Despite the continuing progress in dataflow analysis research, its widespread use in real-world tools is hindered by several serious challenges. One of the central problems is the underlying model of analysis assumed in most of the work in this area. The essence of this model is the assumption of a *whole-program analysis for a homogeneous program*. Interprocedural whole-program analysis takes as input an entire program and produces information about the possible run-time behavior of that program. A fundamental assumption of this analysis model is that the source code for the whole program in available for analysis. Furthermore, such analysis typically treats the entire program as a homogeneous entity, and does not take into account the program's modular structure.

Modern software systems have characteristics that present critical challenges for this traditional model of dataflow analysis. In particular, systems often incorporate *reusable components*. Whole-program analysis is based on the implicit assumption that it is appropriate and desirable to analyze the source code of the entire program as a single unit. However, this assumption is clearly violated for software systems that are built with reusable components:

- Some program components may be available only in binary form, without source code, which makes whole-program analysis impossible.
- Analysis results may be needed even when a whole program simply does not exist. For example, when the developer of a component wants to use a dataflow analysis for understanding, restructuring, or testing of her code, she often does this without having all other components that will eventually be combined to construct a complete application.
- From the point of view of analysis running time and memory usage, it is highly undesirable to reanalyze a component every time this component is used as part of a new system. For example, a popular library may be used in thousands of different applications, and whole-program analysis requires reanalysis of this library from scratch as part of each such application.
- Treating the program as a homogeneous entity means that code changes in one component typically require complete reanalysis of the entire application.
- The running time of whole-program analysis is often dominated by the analysis of the underlying large library components. Thus, to achieve practical running times, analyses often must employ approximations which typically reduce the precision and usefulness of the analysis solution.

The essence of the problem is the following: whole-program interprocedural dataflow analysis is often *impossible* or *inefficient* for software systems that employ reusable components. Thus, the real-world usefulness of hundreds of existing analyses remains questionable. In many case these analyses cannot be used at all. Even if they are possible, they have to be relatively approximate and imprecise in order to scale for industrial-sized software containing hundreds of thousands lines of code in multiple components. As a result of these approximations, the precision of the computed information usually suffers, which leads to problems such as spurious dependencies in program understanding tools, false warnings in verification tools, and infeasible test coverage requirements in testing tools. Such problems ultimately reduce the productivity of the software developers and testers that are clients of these tools.

Our work defines an alternative conceptual model of dataflow analysis which we refer to as *component-level analysis* (CLA). While a whole-program analysis takes as input the source code of a complete program, a component-level analysis processes the source code of a single program component, given some information about the environment of this component. This conceptual model presents a starting point for dataflow analyses that can be applied to software systems built with reusable components.

Given the CLA model, there are two major challenges that static analysis researchers need to address. First, what is the appropriate theoretical framework for designing CLA? Whole-program analyses can be defined in terms of a dataflow lattice, a set of dataflow functions, and algorithms that use the lattice and the functions to compute a dataflow solution [1]. How should this general framework be extended to handle CLA? Second, given the theoretical foundations for CLA, how should specific whole-program analyses be modified to fit the CLA model? For example, how should researchers and tool builders define

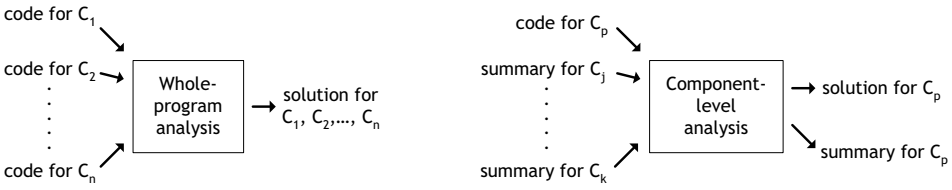

Fig. 1. Whole-program analysis vs. component-level analysis

CLA versions of popular analyses such as pointer analysis, side-effect analysis, data dependence analysis, and constant propagation analysis?

The program analysis research community has been slow to respond to these challenges. For example, at present there does not exist a general theoretical framework for designing CLA. There have been several efforts to define CLA versions of specific kinds of analyses (due to space constraints, we do not discuss these efforts). However, this existing work has been somewhat sporadic and ad hoc in nature, and it has not addressed some of the key technical problems in this domain. The goal of this paper is to describe our initial work on defining the theoretical foundations of CLA. We consider these preliminary results to be a first step in a long-term research agenda aimed at dataflow analysis techniques that can be used for component-based software systems.

2 Component-Level Analysis

The limitations of whole-program dataflow analysis can be addressed by a conceptually different model for interprocedural dataflow analysis: *component-level analysis*. The differences between whole-program analysis and component-level analysis are illustrated in Fig. 1. Whole-program analysis takes as input the source code for all components in a complete program, and produces information about that entire program. Component-level analysis considers a particular component under analysis C_p that is built on top of existing components $C_j, \ldots, C_k$. The analysis input is the source code of C_p together with summary information about the rest of the components. The *summary information* is component metadata that encodes properties of $C_j, \ldots, C_k$ which are relevant for the analysis of C_p. This information can be enclosed with the binary code for the components, allowing analysis of C_p without having access to the source code of the other components. The analysis outputs a component-level solution for C_p and summary information for C_p.

The *component-level solution for* C_p represents properties of the possible behavior of C_p when it interacts with the existing components and with other unknown components. This solution should be computed with *conservative assumptions* about the behavior of components that have not been built yet, and about existing components for which summary information and source code are not available. Thus, the availability of summary information for other components determines the degree of conservativeness in the solution for C_p.

The *summary information for* C_p encodes properties that are relevant for subsequent analysis of components built on top of C_p. This information could be created either independently of the summaries for $C_j, \ldots, C_k$ (in which case it encodes properties of C_p that are independent of any other components), or it could be based on these summaries (and therefore describes properties of C_p in the context of $C_j, \ldots, C_k$). The summary information should be *precise* in the sense of allowing subsequent component-level analysis to be as precise as whole-program analysis would have been. For example, consider a complete program with two components: a library component *Lib* and a main component *Main*. If a summary for *Lib* is included together with the library binary, subsequent component-level analysis of *Main* should produce a solution which is equivalent to the solution for *Main* that would have been computed by a whole-program analysis which analyzed the source code of *Lib* and *Main* together.

Component-level analysis goes beyond the limitations of whole-program analysis because it allows: (1) analysis of a component without the source code of related components, by using the summary information for these components; (2) analysis in the absence of any summary information, by employing conservative assumptions; (3) reuse of the summary information in order to avoid repeated reanalysis of a component; and (4) reduced work to handle code changes, since components that are not affected by the changes do not need to be reanalyzed.

3 Theoretical Foundations

A key challenge for the widespread adoption of the analysis model described in the previous section is the lack of comprehensive theoretical foundations for such analysis. This motivates our ongoing work on defining a general theoretical framework for component-level analysis. This paper outlines initial work on defining a framework for one simple instantiation of the general CLA model. We consider analysis of a complete program that is built with two components: a library component *Lib* and a main component *Main*. The problem under consideration is to precompute summary information for *Lib* and to use it in a subsequent component-level analysis of *Main*. Even this simplified version of the model has important practical implications. For example, there are extensive standard libraries that are associated with languages such as C++, Java, and C#. A standard library could be considered as component *Lib*, while a program written on top of the library is component *Main*.

Our work makes some additional simplifying assumptions. First, the definition of a component is simply "a set of related procedures or classes". The interactions between components are exclusively through calls to methods and procedures and through accesses to shared variables. Clearly, our future work must consider more sophisticated component models; nevertheless, we believe that the simple model used in this paper is the appropriate starting point for this investigation. Another assumption is that the analysis takes into account only the source code and the summary information that is constructed from this source code. If, for example, a formal specification is available for a component

C, it may be possible to utilize this information when analyzing other components that interact with C. Traditional whole-program analysis does not take into account information from formal specifications, but it is clear that future work on component-level analysis should investigate this possibility.

In the standard formulation [1], a whole-program interprocedural dataflow analysis constructs a tuple $\langle G, L, F, M, \eta \rangle$ where

- $G = (N, E)$ is an interprocedural control-flow graph in which each node represents a program statement and each edge represents potential flow of control. An edge (n_1, n_2) is *intraprocedural* when both n_1 and n_2 belong to the same procedure/method. *Interprocedural* edges connect a call node with the entry and exit nodes of the called procedure/method.
- L is a meet semi-lattice whose elements encode program properties. L is a partially ordered set with a partial order $\sqsubseteq$. Intuitively, if $l_1 \sqsubseteq l_2$, the property encoded by l_1 "subsumes" the property encoded by l_2. For each pair $l_1, l_2 \in L$, there exists a unique element $l \in L$ which is the "meet" of l_1 and l_2: $l = l_1 \sqcap l_2$. In essence, l represents a property which is the "merge" of the two properties encoded by l_1 and l_2.
- $F \subseteq \{f \mid f : L \to L\}$ is a monotone function space that is closed under functional composition and functional meet.
- $M : E \to F$ is an assignment of functions to graph edges. The dataflow function $f_e = M(e)$ encodes the effects of e's execution: if property $l \in L$ holds immediately before e, property $f_e(l)$ holds immediately after e.
- $\eta \in L$ is the dataflow solution at the start node of the program.

A path in G is a sequence of edges $p = (e_1, \ldots, e_k)$ such that the target of e_i is the same as the source of e_{i+1}. The dataflow function associated with p is the composition of the functions for the edges: $f_p = f_{e_1} \circ \ldots \circ f_{e_k}$. A *valid path* in G is a path on which calls and return are properly matched. That is, whenever a valid path contains an edge e from the exit node of a method/procedure m to some call node c, the last unmatched call-to-entry edge that precedes e in the path is an edge from c to the entry of m. A dataflow analysis algorithm defines an approach for computing a solution $S_n \in L$ at every node n such that $S_n \sqsubseteq f_p(\eta)$ for each path p from the start node of the program to n. This solution is guaranteed to represent all properties that may hold at n during run-time execution.

3.1 Restricted Component-Level Analysis

Consider again a complete program with two components *Lib* and *Main*. Suppose that this program does not use callbacks that allow the library to call back the main component. For example, in C and C++, the library does not make indirect calls through function pointers that point to functions defined in *Main*. Similarly, in C++ and Java, this means that the library does not make virtual calls that could be resolved through virtual dispatch to methods that are defined in *Main* and override existing methods from *Lib*.

Under this constraint, the classical theoretical framework for whole-program analysis can be adapted for component-level analysis. For each library method or

procedure m that may be called by some main component, a *summary function* f_m can be precomputed in advance. This computation uses only the source code of *Lib* and is independent of *Main*. The set of summary functions constitutes the summary information for *Lib*. A subsequent component-level analysis of *Main* uses these f_m to model the effects of library calls being made by the main component. The resulting solution of the analysis of *Main* can be guaranteed to be as precise (with respect to *Main*) as the solution what would have been computed if a whole-program analysis of $Main \cup Lib$ were performed.

The summary function f_m for a library method/procedure m considers all valid paths that begin at the start node of m, end at the exit node of m, and contain an equal number of calls and returns. Each such path p represents a runtime execution from the moment when m is called to the moment m returns to its caller. Since there are no callbacks from *Lib* to *Main*, each such p stays entirely inside the library component. The summary function for m is $f_m = f_{p_1} \hat{\sqcap} \ldots \hat{\sqcap} f_{p_k}$ for all such paths p_i; this function captures all possible effects of calls to m. Here $\hat{\sqcap}$ denotes the generalization of $\sqcap$ to functions: $(f_1 \hat{\sqcap} f_2)(l) = f_1(l) \sqcap f_2(l)$ for any $l \in L$. When component-level analysis of some *Main* is performed, a call to m is modeled by applying f_m to the current lattice element at the call site.

For brevity, we do not discuss the numerous technical details related to this approach. The key observation is that this form of analysis is a natural generalization of a well-known traditional technique for whole-program analysis (the "functional approach" from [1]). Some existing work on specific analyses uses this approach, either explicitly or in spirit. The success of this technique depends primarily on having compact representations of dataflow functions and on inexpensive compositions and meets of such functions. This issue has been resolved for a broad range of popular analyses [2, 3], but some open questions remain for certain classes of dataflow problems (e.g., non-distributive problems).

4 Generalized Component-Level Analysis

The approach described in Sect. 3.1 is based on a key assumption: the absence of callbacks from *Lib* to *Main*. However, callbacks are quite common in real-world software. Function pointers in C are often used for extending and customizing library functionality. For example, in order to allow a library function `f` to behave in a polymorphic manner, one of the formal parameters of `f` could be a function pointer `g` to a callback function defined in the main component. An indirect call `(*g)(..)` inside `f` invokes the callback function. The complete behavior of `f` is not known to a static analysis until after the library is combined with *Main*. Thus, it is impossible to construct a summary function for `f` using the approach described in Sect. 3.1. Existing work indicates that callbacks through function pointers are used extensively in real-world C libraries [4].

Callbacks occur naturally in object-oriented software. Consider a library method `m(A *a)` in C++ or `m(A a)` in Java, where `A` is a library class. Suppose some *Main* creates a subclass `B` of `A` that overrides some of `A`'s methods. If *Main* calls `m` with an actual parameter that is a pointer to an instance of `B`, a

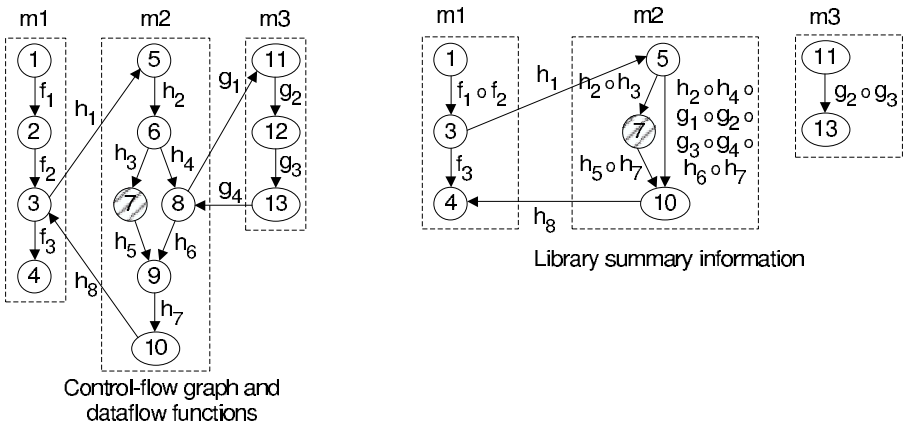

Fig. 2. Summary information in the presence of callbacks.

virtual call through `a` inside `m` may invoke a method defined in `B`. In fact, this is the standard extensibility mechanism for object-oriented libraries. For example, a study from [5] shows that in several packages from the standard Java libraries, typically at least 5% of the virial call sites potentially invoke callback methods, and in some cases the percentage is higher than 30%. In the presence of overriding methods in the main component, it is impossible to use the approach from Sect. 3.1 to create summary information about an object-oriented library.

To address this problem, we propose a new approach for component-level analysis. The preanalysis of *Lib* constructs detailed information which is subsequently combined with the source code of *Main* to compute information that is as precise as the solution that would have been computed for *Main* by a whole-program analysis. For brevity, we present the approach through the example in Fig. 2 rather than describing the underlying formalism. The first part of the figure shows a library component with a set of methods/procedures $\mathcal{P} = \{m1, m2, m3\}$. The control-flow graph is shown together with the dataflow functions for graph edges. Assume that each element of $\mathcal{P}$ could be invoked by code in future main components. The second part of the figure shows the summary information constructed by our approach.

The generation of the summary information starts by identifying call nodes inside *Lib* that could call back some code in some future *Main*. Various techniques can be used to identify such call sites (e.g., conservative forms of pointer analysis); examples of such techniques are available in [6, 5]. In Fig. 2, node 7 represents a callback site. A method or procedure $m \in \mathcal{P}$ is *incomplete* if m contains a callback site or if m invokes (directly or transitively) some other library method m' which contains a callback site. In Fig. 2, $m1$ and $m2$ are incomplete. If m is incomplete, it is impossible to represent the complete effects of calls to m during the preanalysis of the library.

The summary information for *Lib* contains a *reduced control-flow graph* $G' = (N', E')$. The node set N' is a subset of the node set in the "standard" control-

flow graph G. Nodes in N' correspond to (1) start nodes and exit nodes of all incomplete $m \in \mathcal{P}$, (2) all call nodes that are potential callback sites, and (3) all call nodes that invoke incomplete $m \in \mathcal{P}$. Edges in E' represent *paths* in the standard graph G. Consider two nodes $n_1, n_2 \in N'$ such that both belong to the same $m \in \mathcal{P}$. The paths represented by an edge $(n_1, n_2) \in E'$ correspond to sequences of execution steps for which the preanalysis of the library has complete knowledge. For example, in Fig. 2, edge $(5, 10)$ represent the run-time behavior of a call to $m2$ during which the callback site 7 is *not* executed, and therefore all necessary information for the construction of a summary function is available. On the other hand, since the execution of 7 invokes unknown code, this node has to be preserved in the summary information. In essence, the summary functions capture the (incomplete) knowledge that can be inferred "locally" from the library source code. When the main component eventually becomes available, the standard dataflow-analysis techniques can be applied on the control-flow graph of *Main* combined with the reduced control-flow graph from the library.

5 Future Work

We are currently working on the detailed formulation of the proposed technique, and on applying it to specific analyses for Java software (e.g., pointer analysis and dependence analysis). In this context the standard Java libraries are component *Lib*, and the user program is component *Main*. Since the standard libraries contain thousands of classes, they present a challenging scalability problem for whole-program dataflow analyses. Our near-term goal is to demonstrate, both theoretically and experimentally, a set of techniques that allow practical and precise dataflow analysis of real-world Java software. A more general long-term goal is to consider various component models and the appropriate theoretical foundations for dataflow analyses in the context of these models.

References

1. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications. Prentice Hall (1981) 189–234
2. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: ACM SIGPLAN Symposium on Principles of Programming Languages. (1995) 49–61
3. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science **167** (1996) 131–170
4. Milanova, A., Rountev, A., Ryder, B.G.: Precise call graphs for C programs with function pointers. International Journal of Automated Software Engineering **11** (2004) 7–26
5. Kuck, F.C.: Class analysis for extensible Java software. Master's thesis, Ohio State University (2004)
6. Rountev, A., Ryder, B.G., Landi, W.: Data-flow analysis of program fragments. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering. LNCS 1687 (1999) 235–252