Points-to Analysis of RMI-based Java Programs

Yogesh Pingle
M.E.(Comp.), Lecturer, Dept. of Information Technology, VCET
yogesh_pingle@yahoo.com

Pallavi Satpute and Sandesh Patil
B.E.(INFT), Mumbai University
pallavi_16387@yahoo.co.in
sandesh.patil@rediffmail.com

ABSTRACT
Distributed applications play an important role in various commercial, scientific, and engineering domains. The development of such applications poses numerous problems related to software correctness, performance, and maintainability. This paper focuses on distributed Java programs built with the help of the Remote Method Invocation (RMI) mechanism. We consider points-to analysis for such applications. Points-to analysis determines the objects pointed to by a reference variable or a reference object field. Such information plays a fundamental role as a prerequisite for many other static analyses. We present an algorithm for implementing a flow- and context-insensitive point-to analysis for such applications. The work described in this paper solves one key problem for static analysis of RMI programs and provides a starting point for future work on improving the understanding, testing, verification, and performance of RMI-based software.

KEYWORDS
RMI, object-oriented software, distributed software, points-to analysis, call graph construction.

1 INTRODUCTION
Java Remote Method Invocation (RMI) is an object model for developing distributed applications in Java [3]. Using RMI, objects in one Java virtual machine (JVM) can invoke methods on objects in other JVMs. RMI provides powerful features such as object references that cross JVM boundaries, remote invocations that can use entire object graphs as parameters, and distributed garbage collection. RMI can be used either as a stand-alone middleware platform, or a foundation for more advanced architectures.

Distributed applications play an important role in various commercial, scientific, and engineering domains. The development of such applications poses numerous problems related to software correctness, performance, and maintainability. The target of our work is points-to analysis [1]. Such analysis determines the objects to which local variables, formal parameters, and fields may point. The problem of developing precise and practical points-to analyses is of great importance for static analysis researchers, as well as for builders of tools for software understanding, testing, and static checking. Points-to information plays an important enabling role for a large number of static program analyses. Experience has shown that a points-to analysis is often a critical prerequisite for a variety of program understanding applications, testing approaches, software verification techniques, and performance optimizations.

Our first goal is to establish the foundations for points-to analysis of RMI-based Java applications. These will provide a basis for defining a wide range of points-to analyses for RMI applications and they will enable work on RMI-based extensions of other popular static analyses. Our second goal is to define an algorithm for implementing the points-to analysis. The algorithm is a generalization of an approach by Lhotak and Hendren [4] for nondistributed Java programs.

2 RELATED WORK
There are several algorithmic dimensions [1] that affect the cost/precision trade-offs of points-to analyses. The most important dimensions are as follows:

- **Flow Sensitivity.** Flow-insensitive analyses do not take into account the flow of control within a procedure; they compute a single set of points-to relationships for the entire procedure. Flow-sensitive analyses compute a separate solution at each program point inside a procedure.

- **Context Sensitivity.** Context-insensitive analyses do not attempt to distinguish among the different invocation contexts of a procedure. Context-sensitive analyses employ some abstraction of the calling context; as a result, such analyses are potentially more precise and more expensive than context-insensitive ones.

- **Field Sensitivity.** A field-sensitive analysis computes a separate point-to solution for each field of each abstract object. A field-insensitive analysis computes a points-to solution for the entire abstract object.

- **Directionality.** An equality-based analysis treats an assignment \( p = q \) as representing a bidirectional flow of values from \( p \) to \( q \) and from \( q \) to \( p \). A subset-based analysis treats such an assignment as a unidirectional flow of values from \( q \) to \( p \).

- **Call Graph Construction.** Some analyses use a precomputed conservative call graph, while others
compute the call graph on the fly during the analysis, as points-to relationships are discovered.

3 OVERVIEW OF RUNNING EXAMPLE

The goal of this section is to introduce key RMI concepts that are relevant for the points-to analysis described in the next section. The input to the analysis contains the code for several components \( C_1, C_2, \ldots, C_k \). The set of components will be denoted by \( C \). For each component \( C_i \in C \), the analysis takes as input a set \( cls(C_i) = \{ X_1, \ldots, X_{n_i} \} \) of Java classes.

Fig. 1 and 2 show the example used in the rest of the paper; this example is taken from [1]. For simplicity, we exclude error-handling code. The example contains events, listeners for these events, channels along which events are announced to the listeners, and event sources that create the events and send them to the channels.

In \( C_1 \), at lines 22-23 in Fig. 2, \( MyChannel \).main creates an instance of remote class \( MyChannel \) and registers it with a naming service. In \( C_2 \), \( MyListener \).main uses the naming service to obtain a reference to the remote channel object (line 29 in Fig. 2), and then registers with the channel two remote listener objects (lines 30-31). Similarly, in \( C_3 \), \( EventSource \).main obtains a reference to the remote channel object (line 35) and then announces an event on the channel (line 36). In \( MyChannel \).announce, the channel object dispatches the event to the registered remote listeners (line 19).

For each pair of components \( (C_i, C_j) \in C \times C \), the analysis input contains a set \( I_{i \rightarrow j} \) of pairs of local variables. Each pair \( (v_1, v_2) \) represents a use of the external mechanism, which results in creating remote references from \( v_2 \) in \( C_j \) to all remote objects pointed-to by \( v_1 \) in \( C_i \). For our running example, \( I_{1 \rightarrow 2} = \{ (e, f) \} \) due to lines 23 and 29, and \( I_{1 \rightarrow 3} = \{ (e, h) \} \) due to lines 23 and 35.

4 ANALYSIS ALGORITHM

This section describes an algorithm [1] for implementing the points-to analysis.

4.1 Variable Names and Object Names

The analysis can be defined in terms of several sets [1]. Let \( cls \) be the union of all sets of classes \( cls(C_i) \) for all components \( C_i \). We will denote by \( L \) the set of all local variables, formal parameters, and implicit parameters this in \( cls \). Similarly, let \( F \) and \( SF \) be the sets of all instance fields and static fields in \( cls \), respectively. Finally, let \( S \) be the set of all allocation expressions of the form new \( X(...) \) in \( cls \).

For each pair of components \( (C_i, C_j) \in C \times C \), the analysis input contains a set \( I_{i \rightarrow j} \) of pairs of local variables. Each pair \( (v_1, v_2) \) represents a use of the external mechanism, which results in creating remote references from \( v_2 \) in \( C_j \) to all remote objects pointed-to by \( v_1 \) in \( C_i \). For our running example, \( I_{1 \rightarrow 2} = \{ (e, f) \} \) due to lines 23 and 29, and \( I_{1 \rightarrow 3} = \{ (e, h) \} \) due to lines 23 and 35.

Fig. 1. Running example, part 1.

Fig. 2. Running Example, part 2.
entire application, as well as component-specific sets of reachable methods \( \text{Reach}_i \) for all \( C_i \in C \).

### 4.2 Pointer Assignment Graph

The analysis algorithm uses a data structure referred to as a pointer assignment graph (PAG) [1]. Nodes in this graph represent memory locations or expressions that refer to such locations. The edges of the graph represent the flow of information between the nodes. For example, if a statement \( v_1 = v_2 \) belongs to some method from \( \text{Reach}_i \), the PAG contains an edge \( \text{node}(v_2) \rightarrow \text{node}(v_1) \).

### 4.3 Points-to Graph

The analysis builds a points-to graph [1] in which the edges represent points-to relationships. An edge \((v_i, o) \in V \times O\) shows that a variable represented by \( v_i \) may point to an object represented by \( o \). An edge \((v_i, o)\) could be either a remote edge, denoted by \((v_i, o)\_R\), or a local edge, denoted by \((v_i, o)\_L\). Fig. 3 shows some of the variable names and object names for the running example, as well as several points-to edges. Edges labeled with [ ] represent points-to relationships for array elements.

![Fig. 3. Partial points-to graph for the running example.](image)

### 4.4 Algorithm Overview

Given the source code for all classes in sets \( \text{cls}(C_i) \) for all program components \( C_i \in C \), the algorithm produces:

- A pointer assignment graph (PAG).
- Local points-to set \( \text{Pt}_L \) and remote points-to set \( \text{Pt}_R \) for PAG nodes \( \text{node}(v_i) \) and \( \text{node}(o.fld) \).
- A set of reachable methods \( \text{Reach} = \bigcup_i \text{Reach}_i \), where \( m_i \in \text{Reach}_i \) represents the copy of method \( m \) in component \( C_i \).
- A call graph with nodes \( m_i \in \text{Reach} \) and edges \( e \in \text{Reach} \times \text{Reach} \times \text{Call Sites} \times \{L, R\} \).

A remote call graph edge \((m_i', n', c)\_R\) indicates that method \( m \) in component \( C_i \) contains a call site \( c \) at which one possible remote runtime target method is \( n \) in component \( C_j \). A local call graph edge \((m_i', n', c)\_L\) shows that method \( m \) in \( C_i \) contains a call site \( c \) at which one possible nonremote runtime target method is \( n \) in the same component.

The algorithm is a generalization of the algorithm from [4], [5] for non-RMI Java. New techniques are introduced by [1] in order to handle remote references, remote calls, and serialization. For simplicity, we have omitted some part of the algorithm which had no effect on the output of the running example used in this paper. The computation is based on a worklist of PAG nodes whose local or remote points-to sets have changed. When a worklist element is processed, new elements are added to the points-to sets of other PAG nodes. The propagation can also result in 1) finding new reachable methods, 2) creating new PAG edges for actual-formal parameter pairs and for method return values, and 3) creating new remote PAG edges to represent the effects of serialization. New reachable methods and new PAG edges trigger additional propagation. The process continues until no additional information can be inferred.

![Fig. 4. Top level of the analysis algorithm.](image)
such that one variable is local to \( m' \) and the other one is local to \( n' \). The analysis creates special PAG edges between these locals; these edges are labeled “remote”. The source nodes of the new PAG edges are added to NodeWorklist, because the points-to sets of these nodes may have to be propagated to the corresponding target nodes.

Finally, at lines 16-17, for all compile-time monomorphic call sites in \( m' \), the corresponding call graph edges are created using procedure \textit{AddCallGraphEdge}.

Procedure \textit{AddCallGraphEdge}, shown in Fig. 6, is invoked when a potential target method is detected at a call site. The procedure updates the call graph, the set of reachable methods, and the PAG. Lines 21-25 in Fig. 6 create new PAG edges representing the flow of values due to actual-formal parameter bindings and due to return values. The newly created PAG edges are remote if the call graph edge is remote and local otherwise. The source nodes of the new edges are added to NodeWorklist.

Procedure \textit{PropagatePointsToSets} in Fig. 8 contains the main loop of the algorithm. This loop propagates objects to points-to sets and performs on-the-fly call graph construction. The procedure completes when there are no more objects to be propagated to points-to sets.

Each iteration of the inner loop (lines 27-35) processes a PAG node \( \text{node}(v') \). This node was put on \( \text{NodeWorklist} \) earlier because the (local or remote) points-to set of \( v' \) changed, and this change required further propagation.

After all necessary changes to the call graph have occurred, the PAG edges related to \( v' \) are examined and the corresponding points-to sets are updated (lines 30-33).
5 CONCLUSION

It is possible to naturally generalize the existing formalisms for points-to-analysis to handle RMI features such as remote references, remote calls, and parameter passing through serialization. The key to this generalization is to maintain two separate points-to sets per variable: one for ordinary references and one for remote references. The overall conclusions from the experimental study are the following: First, the analysis appears to achieve high precision when modeling the semantics of remote calls. Second, the analysis suffers from the same problem exhibited by subset-based points-to analysis for non-distributed Java programs: most of the running time is spent in the standard libraries.

REFERENCES