Static Analysis of Object References in RMI-Based Java Software

Mariana Sharp, *Student Member*, *IEEE Computer Society*, and
Atanas Rountev, *Member*, *IEEE Computer Society*

Abstract—Distributed applications provide numerous advantages related to software performance, reliability, interoperability, and extensibility. 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 the first theoretical definition of points-to analysis for RMI-based Java applications, and we present an algorithm for implementing a flow- and context-insensitive points-to analysis for such applications. We also discuss the use of points-to information for computing call graph information, for understanding data dependencies due to remote memory locations, and for identifying opportunities for improving the performance of object serialization at remote calls. 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.

Index Terms—RMI, object-oriented software, distributed software, program analysis, points-to analysis, reference analysis, class analysis, call graph construction, side-effect analysis.

1 Introduction

Java Remote Method Invocation (RMI) is an object model for developing distributed applications in Java [26]. 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. For example, both Enterprise JavaBeans and Jini are based on RMI and also provide additional middleware services.

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. For RMI applications in particular, some approaches have been investigated for program understanding, performance optimizations, and software testing (e.g., [31], [30], [16], [25], [11], [50], [32]). However, at present there is no work on establishing *systematic foundations for static analysis* of RMI applications. The goal of this paper is to take a significant step toward defining such foundations.

The target of our work is *points-to analysis*. Such analysis determines the objects to which local variables, formal parameters, and fields may point. As evidenced by the extensive existing work on points-to analysis for C and Java, 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. There has been a large body of work on points-to analysis; a brief overview of the relevant analyses is presented in the next section. However, these existing analyses cannot be applied directly to RMI-based distributed Java applications. Thus, the builders of such applications cannot take advantage of a large number of well-known static analyses—both points-to analyses as well as other popular analyses that require points-to information. Developing high-quality points-to analyses for RMI applications is an essential first step for designing a wide range of software tools for such applications (e.g., change-impact tools and test-coverage tools).

Theoretical Model. Our first goal is to establish the foundations for points-to analysis of RMI-based Java applications. The importance of these foundations is twofold. First, they provide a basis for defining a wide range of points-to analyses for RMI applications, based on the large number of such analyses for nondistributed programs. Second, they enable work on RMI-based extensions of other popular static analyses (e.g., dependence analyses, side-effect analyses, program slicing, change impact analyses, etc.).

Analysis Algorithm. Our second goal is to define an algorithm for implementing the points-to analysis. The

- *The authors are with the Department of Computer Science and Engineering, Ohio State University, Columbus, OH 43210. E-mail: {sharpm, rountev}@cse.ohio-state.edu.*

Manuscript received 12 Dec. 2005; revised 9 Apr. 2006; accepted 20 July 2006; published online 27 Sept. 2006.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0328-1205.

algorithm is a generalization of an approach by Lhoták and Hendren [21] for nondistributed Java programs. We introduce new techniques that allow the analysis to represent the flow of remote object references, the effects of remote invocations, and the remote propagation of object graphs through serialization. Furthermore, we present an approach for efficient modeling of the code in the standard Java libraries; our experiments indicate that this approach is essential for reducing the running time of the analysis.

Static Analyses for Program Understanding. Another goal of this work is to describe three uses of points-to analysis for the purpose of understanding RMI applications. First, we outline the use of the computed call graph to answer questions related to the intermethod and intercomponent flow of control. Second, we outline the use of points-to information to identify write-read dependencies due to remote calls. In particular, we consider *intercomponent* dependencies, in which components running in two different JVMs potentially access the same memory location. Third, we discuss the use of the points-to analysis to identify opportunities for improving the analyzed program by reducing the cost of serialization at remote calls [50].

Analysis Implementation. Our fourth goal is to implement and evaluate the points-to analysis. We present a preliminary experimental study on a set of 12 RMI applications. Our initial results suggest that the analysis could be a good candidate for a general-purpose points-to analysis of RMI-based programs.

2 Related Work

The analysis of points-to relationships is a research problem that has received a lot of attention in the last 15 years, due to the important role it plays as an enabling technology for a variety of other static analyses. This section provides an overview of some of the related work in this field; detailed surveys are available in [18], [40]. There are several algorithmic dimensions 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 (or a method); they compute a single set of points-to relationships for the entire procedure. In contrast, flow-sensitive analyses compute a separate solution at each program point inside a procedure. Theoretically, flow-sensitive analyses are more expensive and more precise than their flow-insensitive counterparts.
- *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. In *parameter-based* context-sensitive analyses, calling context is modeled by using some abstraction of the values of the actual parameters at a call site, while *call-chain-based* context-sensitive analyses represent the context with a vector of call sites for the procedures that are currently active on the runtime call stack.
- *Representation for Dynamically-Allocated Objects.* There are various possible representations for entities that are allocated dynamically—for example, due to `malloc` calls in C or `new` expressions in C++ and Java. For analysis of C, typically a separate abstract object (i.e., an analysis abstraction of a set of runtime objects) is used for each `malloc` call site. For analysis of object-oriented programs, the two most popular schemes are to use 1) one abstract object per class, or 2) one abstract object per `new` expression. More precise versions of these schemes have also been proposed—for example, by introducing context-sensitive versions of abstract objects.
- *Field Sensitivity.* A field-sensitive analysis computes a separate points-to solution for each field (e.g., object field in C++/Java or structure field in C/C++) of each abstract object. A field-insensitive analysis computes a points-to solution for the entire abstract object, without distinguishing the values of individual fields. Some points-to analyses for object-oriented languages are field-based, meaning that a separate solution is computed for each declared field without distinguishing the incarnations of this field across different abstract objects.
- *Directionality.* This dimension typically applies to flow-insensitive analyses. 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`—that is, the points-to solutions for `p` and `q` are considered to be the same. A *subset-based* analysis treats such an assignment as a unidirectional flow of values from `q` to `p`; as a result, the points-to solution for `q` is a subset of the points-to solution for `p`.
- *Call Graph Construction.* The calling structure of the program could depend on points-to relationships—for example, due to function pointer calls in C and polymorphic calls in C++ and Java. 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.

Points-to Analysis for C Programs. Early work on points-to analysis for C programs focused on flow- and context-sensitive algorithms (e.g., [19], [8]). Subsequently, the focus shifted to more scalable flow- and context-insensitive analyses, with examples of equality-based approaches (e.g., [44], [53], [47], [10]), subset-based approaches (e.g., [2]), and combinations of both (e.g., [42], [7]). These analyses were typically field-insensitive, even though some field-sensitive approaches were proposed (e.g., [52]). A category that has received particular attention is flow- and context-insensitive, subset-based, field-insensitive algorithms with on-the-fly call graph construction; such analyses are sometimes referred to as Andersen-style analyses due to the work in [2]. The scalability of such algorithms has been improved dramatically through various techniques (e.g., [9], [46], [35], [17]). In addition to developing different points-to analysis algorithms, researchers have investigated various uses of the points-to

information—for example, for side-effect analysis (e.g., [41]) and for reaching definitions analysis (e.g., [48]).

Points-to Analysis for Java Programs. As with points-to analysis for C, there is a large body of work for points-to analysis for Java, mostly focusing on flow-insensitive algorithms. Typically, these algorithms have been derived from similar algorithms for analysis of C programs. For example, various context-sensitive analyses have been proposed and evaluated (e.g., [15], [5], [39], [6], [28], [51], [29], [24], [22]). Flow- and context-insensitive points-to analyses for Java are considered, for example, in [33], [45], [37], [23], [21], [3]. The closest related work from this category is due to Lhoták and Hendren [21]. They present a family of analyses, the most precise of which is a flow- and context-insensitive, field-sensitive, subset-based analysis that creates one abstract object per `new` expression and uses on-the-fly call graph construction. This particular analysis provides a state-of-the-art implementation of the Andersen-style analyses from [45], [37], [23] and achieves a practical trade-off between cost and precision. This analysis for nondistributed Java applications serves as the starting point for our analysis for RMI-based Java programs. We introduce various modifications of the techniques from [21]. For example, new kinds of analysis propagation rules are required for modeling the behavior of RMI code. The handling of calls is generalized to simulate the semantics of remote invocations, including the effects of serialization of object graphs (Section 4). We also introduce a technique for efficient handling of the standard Java libraries (Section 6).

Points-to Analysis for RMI-based Java Applications. There has been very little work on generalizing points-to analyses to RMI-based Java software. The closest related work is [50], where a compile-time points-to analysis is used to optimize serialization at remote calls. The analysis is described with very little detail, but it appears to be a flow-sensitive and context-insensitive variation of Andersen-style analysis. There is no theoretical definition of the analysis semantics, and no details are given about the algorithms and data structures used to implement this semantics. For example, it is unclear whether the approach uses two different points-to sets (remote and local) per variable, whether component-specific copies v^i of a variable v are used, whether the set of reachable methods is constructed during the analysis or is assumed to be part of the analysis input, and whether the underlying standard libraries are being analyzed. Our work provides a precise theoretical definition as well as specific algorithms and data structures. The experimental results in [50] focus on the effects of the optimizations on performance, while we are primarily interested in uses of the points-to information in tools for software understanding, testing, and verification.

Ahern and Yoshida [1] introduce a Java-like language with RMI mechanisms. Their focus is on defining a formal semantics and typing rules for the language. The type system could potentially provide a formal basis for static analyses of Java RMI applications. However, this theoretical work does not define any specific analysis problems or algorithms and does not perform experiments on actual RMI programs.

```
— Event —
[1] class Event implements Serializable {
[2]  public Date date() { return on; }
[3]  public String description() { return des; }
[4]  public Event(String a) { des=a; on=new Date(); }
[5]  private Date on; private String des; }
— Event Listener —
[6] interface Listener extends Remote {
[7]  public void occurred(Event b); }
— Event Channel —
[8] interface Channel extends Remote {
[9]  public void add(Listener c);
[10] public void announce(Event d); }
```

Fig. 1. Running example, part 1.

3 Overview of Java RMI

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 $\mathcal{C}$. For each component $C_i \in \mathcal{C}$, the analysis takes as input a set $cls(C_i) = \{X_1, \ldots, X_{n_i}\}$ of Java classes. ("Classes" will refer to both Java classes and Java interfaces.) Each component is executed in a separate JVM, typically on a different physical machine. Set $cls(C_i)$ is the complete set of classes that may be loaded at runtime in the JVM that executes component C_i. Note that an implementation of the RMI mechanism requires additional helper classes that are generated automatically from classes in $cls(C_i)$. For example, in the default implementation of RMI by Sun, the `rmic` compiler produces a variety of stub classes that implement the details of remote invocations. Such classes are not part of the analysis input.

For any two components C_i and C_j, sets $cls(C_i)$ and $cls(C_j)$ are not necessarily disjoint: It is possible for the same class to be loaded in the two virtual machines that execute C_i and C_j. One example is the classes from the standard Java libraries. We assume that the same version of the libraries is loaded in each JVM; thus, all library classes are included implicitly in $cls(C_i)$ for all $C_i \in \mathcal{C}$.

3.1 Running Example

Figs. 1 and 2 show the example used in the rest of the paper; this example is based on a similar example from [13]. For simplicity, we exclude error-handling code (e.g., code related to exceptions thrown by remote invocations). 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. We consider the following configuration of components:

$$cls(C_1) = \{\texttt{Event}, \texttt{Listener}, \texttt{Channel}, \texttt{MyChannel}\},$$
$$cls(C_2) = \{\texttt{Event}, \texttt{Listener}, \texttt{Channel}, \texttt{MyListener}\},$$
$$cls(C_3) = \{\texttt{Event}, \texttt{Listener}, \texttt{Channel}, \texttt{EventSource}\}.$$

```
—— Event Channel Implementation: Component C1 ——
[11] class MyChannel implements Channel
[12]                 extends UnicastRemoteObject {
[13]  private Listener[] all; private int num;
[14]  public MyChannel() {
[15]   Listener[] arr = new Listener[10];
[16]   all = arr; num = 0; }
[17]  public void add(Listener c) { all[num++] = c; }
[18]  public void announce(Event d) {
[19]   for(int i=0; i<num; i++) all[i].occurred(d); }
[20]  public static void main(String[] args) {
[21]   String channel_id = args[0];
[22]   Channel e = new MyChannel();
[23]   Naming.bind(channel_id,e); } }
—— Event Listener Implementation: Component C2 ——
[24] class MyListener implements Listener
[25]                  extends UnicastRemoteObject {
[26]  public void occurred(Event b) {...}
[27]  public static void main(String[] args) {
[28]   String channel_id = args[0];
[29]   Channel f = (Channel)Naming.lookup(channel_id);
[30]   Listener g = new MyListener(); f.add(g);
[31]   g = new MyListener(); f.add(g); } }
—— Event Source Implementation: Component C3 ——
[32] class EventSource {
[33]  public static void main(String[] args) {
[34]   String channel_id = args[0];
[35]   Channel h = (Channel)Naming.lookup(channel_id);
[36]   Event k = new Event("abc"); h.announce(k); } }
```

Fig. 2. Running example, part 2.

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. (The naming service will be discussed shortly.) 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).

3.2 Basic RMI Concepts

3.2.1 Remote Objects, References, and Calls

A *remote class* implements the marker interface `java.rmi.Remote`; this interface does not contain any methods or fields. A *remote object* is any instance of a remote class. Library class

`java.rmi.server.UnicastRemoteObject`,

which implements `java.rmi.Remote`, provides default support for point-to-point object references using TCP. The simplest mechanism for creating remote classes is to create subclasses of `UnicastRemoteObject`. Other mechanisms are also possible [26], but they are conceptually similar and are beyond the scope of this paper. In the running example, classes `MyChannel` and `MyListener` from Fig. 2 (lines 11 and 24) are remote classes, because they implement `java.rmi.Remote`.

A *remote reference* represents a connection between two different JVMs. Similarly to an ordinary (nonremote) object reference, a remote reference is a pointer to an object. The notion of a remote reference is an abstraction: In reality, a component has a reference to a stub object in its own JVM. Typically, the existence of these stub objects is ignored and, instead, RMI programming uses the abstraction of a reference pointing directly to the remote object. An invocation through a remote reference is a *remote invocation*.

3.2.2 Creation of Remote References

Remote references can be created in several ways. For example, a remote invocation can take as an actual parameter an ordinary reference to a locally created remote object o. As a result of the call, the remotely-invoked method takes as formal parameter a remote reference to o. For example, in Fig. 2, component C_2 creates an ordinary local reference `g` to a newly created object of class `MyListener` (line 30). Due to the remote call to `add` at line 30, the formal parameter `c` of `add` (line 17) contains a remote reference to this instance of `MyListener`.

Another mechanism for obtaining remote references is the use of some *naming service*. The calls to `java.rmi.Naming` in the running example illustrate such use. A naming service is a separate component whose purpose is to allow registration and lookup of remote objects. Sun's RMI implementation provides a default naming service referred to as the *RMI registry*. A call `bind(name,x)` inserts in the registry a reference to the remote object o referred to by `x`, under the given string name. In the running example, the two invocations `lookup(channel_id)` at lines 29 and 35 are used to initialize local variables `f` and `h` with remote references to the remote object of class `MyChannel`.

While the RMI registry provides a simple naming service, in general there could be other mechanisms for establishing initial "bootstrapping" remote references between two components [26]. Here by "bootstrapping" we mean references that are created with the help of some external mechanism (e.g., a naming service) in order to establish initial connections between components. To model such initial references, we assume that the analysis input contains information about the variables through which such references are created. For each pair of components $(C_i, C_j) \in \mathcal{C} \times \mathcal{C}$, the analysis input contains a set $I_{i \to j}$ 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, $I_{1 \to 2} = \{(\mathtt{e}, \mathtt{f})\}$ due to lines 23 and 29, and $I_{1 \to 3} = \{(\mathtt{e}, \mathtt{h})\}$ due to lines 23 and 35. Sets $I_{i \to j}$ depend on the specific mechanism used by the application. It may be possible to construct these sets automatically in some simpler cases (e.g., when using the default RMI registry). However, since, in general, the external mechanism for creating initial remote references could be application-specific, programmer input may be required to obtain the information in $I_{i \to j}$.

3.2.3 Call-by-Copy through Serialization

When actual parameters of a remote call are references to nonremote objects o_i, the parameter passing mechanism used is call-by-deep-copy. Objects o_i together with all other objects reachable from them are subject to serialization. This process encodes the object graph starting from o_i and recreates it in the target JVM. For example, consider the call to `announce` in `EventSource.main` (line 36 in Fig. 2). In this call, the actual parameter is a (nonremote) reference to an instance o of nonremote class `Event`. The class is serializable because it implements interface `java.io.Serializable`. Fields `on` and `des` of o refer to serializable objects (line 4 in Fig. 1), because both `Date` and `String` are serializable classes.

Information about o and the two associated instances of `Date` and `String` is sent across the network. The "mirror image" of this object graph is created in C_1, and formal parameter `d` in `MyChannel.announce` (line 18 in Fig. 2) points to the copy of o. This process does *not* invoke the constructor of `Event` in C_1 on the copy of o. The two calls to `occurred` at line 19 trigger this process again, and in the JVM for C_2 the object graph is recreated twice. Our analysis assumes that objects are serialized using the default serialization mechanism [27] and the application does not use custom serialization methods (e.g., methods such as `writeObject`); this assumption is checked by our implementation.

4 Points-To Analysis

This section defines the theoretical foundations for points-to analysis of RMI-based Java applications. The proposed analysis is subset-based, flow- and context-insensitive, field-sensitive, and builds the call graph on the fly. Our approach could easily be extended to flow- and context-sensitive points-to analyses, and to analyses that are not subset-based. Such extensions are well understood for non-distributed Java programs (e.g., [14], [29]) and there are no conceptual difficulties in defining such extensions for our analysis.

First, we define the basic elements of the analysis: the nodes and edges in the *points-to graphs*. Next, we describe the rules for creating points-to edges, based on the analyzed program code. The formalism presented in this section serves as the basis for the points-to analysis algorithm described in Section 5.

4.1 Variable Names and Object Names

The analysis can be defined in terms of several sets. 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.

The analysis is defined in terms of a set V of *variable names* for reference variables and a set O of *object names* for runtime objects. The set V of variable names is a subset of $(L \cup SF) \times \mathcal{C}$. A pair $(v, C_i) \in V$ represents a local variable, a formal parameter, or a static field v in some class from $cls(C_i)$ such that v exists in the JVM executing C_i. The variable names will be denoted by v^i, where the superscript corresponds to the component. For example, variable `f` in component C_2 (line 29 in Fig. 2) will be denoted by $\mathtt{f}^2$. For the same $v \in L \cup SF$, there potentially may be multiple $v^i \in V$, each one corresponding to a different C_i.

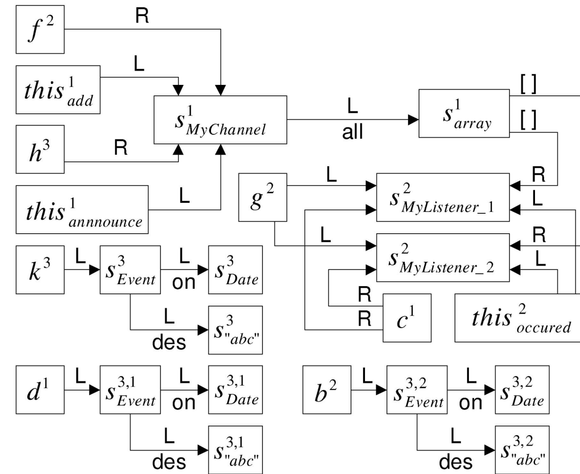

Fig. 3. Partial points-to graph for the running example.

There are two categories of object names $o \in O$. First, $o = (s, C_i) \in S \times \mathcal{C}$ corresponds to runtime objects that are created by object allocation site s when this site is executed in the JVM for component C_i. Each such object is in the address space of that same JVM. Typically, we will use s^i to denote such an object name; as with variable names, the superscript indicates the corresponding component. Each s^i is labeled as remote or nonremote, depending on whether it is an instance of a remote class.

Remote calls can create copies of serializable objects. We use object names $o = (s, C_i, C_j) \in S \times \mathcal{C} \times \mathcal{C}$ to represent such "copy objects." The names will typically be denoted by $s^{i,j}$. Such a name corresponds to a runtime object which exists in the JVM for component C_j and was created as a (transitive) copy of a "normal" object which was created in the JVM for C_i by allocation site s. For example, let s_{Date} be the allocation site `new Date()` in the constructor of `Event` in the running example (line 4 in Fig. 1). Name s^3_{Date} denotes the instance of `Date` which is created in C_3. Due to the remote call to `announce` from C_3 to C_1 (line 36 in Fig. 2), a copy of that `Date` object is created in C_1; the name representing this copy object will be $s^{3,1}_{Date}$. The remote calls to `occurred` from C_1 to C_2 (line 19) create in C_2 two runtime copies of the copy object from C_1. Both objects are transitive copies of the original object from C_3 and are represented by object name $s^{3,2}_{Date}$. Due to the properties of RMI, names $s^{i,j}$ can correspond only to nonremote objects.

4.2 Points-To Graphs

The analysis builds a *points-to graph* 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 $(o_1, f, o_2) \in O \times F \times O$ shows that some object represented by o_1 may store in its f field a reference to an object represented by o_2. An edge (v^i, o) could be either a *remote edge*, denoted by $(v^i, o)_R$ or a *local edge*, denoted by $(v^i, o)_L$. The same subscripts will also be applied to edges (o_1, f, o_2). 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.

For $(v^i, o)_L$, both the variable and the target object must belong to the same JVM. Thus, such edges are either of the form $(v^i, s^i)_L$ or $(v^i, s^{k,i})_L$. For example, in Fig. 3, the two local points-to edges coming out of $\texttt{g}^2$ represent the first case, while the local points-to edge coming out of $\texttt{b}^2$ represents the second case. Note that s^i in $(v^i, s^i)_L$ could be a remote object (i.e., an instance of a class which implements `Remote`), but the reference to it is still an ordinary local reference; again, the two local edges from $\texttt{g}^2$ illustrate this possibility.

Edge $(v^i, s^j)_R$ represents a points-to relationship through a remote reference, and s^j is always a remote object.[1] Since copy objects created due to serialization cannot be remote, it is not possible to have an edge $(v^i, s^{k,j})_R$. For $(o_1, f, o_2)_L$, the two objects belong to the same JVM; either one (or both) could be a copy object $s^{k,i}$ instead of an ordinary object s^i. For example, in Fig. 3, the local edge from s^3_{Event} to s^3_{Date} is between "normal" objects, while the local edge from $s^{3,1}_{Event}$ to $s^{3,1}_{Date}$ is between "copy" objects created due to serialization. For a points-to edge $(o_1, f, o_2)_R$, object o_2 is always a remote object. Fig. 3 shows two examples of such remote edges, both originating from s^1_{array}.

4.3 Elementary Statements

For brevity, we discuss only the following categories of *elementary statements*:

- Direct assignment: $\texttt{v}_1 = \texttt{v}_2$.
- Instance field write: $\texttt{v}_1.\texttt{f} = \texttt{v}_2$.
- Instance field read: $\texttt{v}_1 = \texttt{v}_2.\texttt{f}$.
- Static field write: $\texttt{X.f} = \texttt{v}$.
- Static field read: $\texttt{v} = \texttt{X.f}$.
- Object creation: $\texttt{v} = \texttt{new X}$.
- Static invocation: $\texttt{w} = \texttt{X.m}(\texttt{v}_1, \ldots, \texttt{v}_k)$.
- Instance invocation: $\texttt{w} = \texttt{v}_0.\texttt{m}(\texttt{v}_1, \ldots, \texttt{v}_k)$.

In these statements, $\texttt{v}_i \in L$ denotes a local variable or a formal parameter (including `this`), and `X` denotes a class. As usual in program analysis research, we assume that complex statements and expressions are broken down into the elementary statements listed above, with the help of artificial variables—for example, `x.m1(y.m2())` is represented as `t = y.m2(); x.m1(t)`, where `t` is a helper variable.

The elementary statements above are enough to represent a large subset of Java. Additional language features can be handled as already done in previous work on points-to analysis (e.g., [37], [21]). For example, since our analysis is flow-insensitive, there is no need to consider elementary statements for control-flow features such as branches and loops. As another example, array creation can be handled similarly to a statement $\texttt{v} = \texttt{new X}$. As yet another example, accesses to array elements can be treated similarly to accesses of instance fields—for example, using elementary statements $\texttt{v}_1.[\,] = \texttt{v}_2$ and $\texttt{v}_1 = \texttt{v}_2.[\,]$, where $[\,]$ can be thought of as an artificial field of an artificial class `Array`.

1. It is possible to have $i = j$ and the reference be remote at the same time. For example, if C_i creates a remote object, registers it with a naming service, and then immediately looks it up, the component will obtain a remote reference to the object. Calls through this reference will be remote calls that are handled by the RMI infrastructure.

4.4 Effects of Elementary Statements

The analysis constructs a points-to graph G for the entire application, as well as component-specific sets of reachable methods $Reach_i$ for all $C_i \in \mathcal{C}$. In the beginning, G is empty and each $Reach_i$ contains the main method of the corresponding C_i.[2] For each statement that appears in some method from $Reach_i$ for some i, the analysis adds to G nodes and edges that represent the effects of the statement, and updates all affected sets $Reach_j$. The rules for handling different statements are represented as function definitions of the form $f(G) = G'$, where G and G' are points-to graphs; the discussion below defines nine such rules, **R1** through **R9**.

4.4.1 Lookup of Remote References

The first rule **R1** considers the references that are created with the help of an external mechanism such as a naming service:

R1 for each $(v^i, w^j) \in I_{i \to j}$ such that v^i is a local in some $m' \in Reach_i$ and w^j is a local in some $m'' \in Reach_j$:
$$f(G) = G \cup \{\, (w^j, o)_R \mid (v^i, o)_x \in G \wedge o \text{ is remote} \,\}.$$

Points-to edge $(v^i, o)_x$ could be either local or remote: The object exported by component C_i is either created locally, in which case the edge is $(v^i, o)_L$, or is obtained from some other component, in which case the edge is $(v^i, o)_R$. For example, for the code in Fig. 2, $I_{1\to 2} = \{(\texttt{e}^1, \texttt{f}^2)\}$ and $I_{1\to 3} = \{(\texttt{e}^1, \texttt{h}^3)\}$ due to lines 23, 29, and 35. Since there is a local points-to edge from $\texttt{e}^1$ to $s^1_{MyChannel}$ due to assignment $\texttt{e} = \texttt{new MyChannel}()$ at line 22, the analysis creates remote points-to edges from $\texttt{f}^2$ and $\texttt{h}^3$ to that same object name, as shown in Fig. 3.

4.4.2 Assignments

Suppose the assignment under consideration occurs in some method from $Reach_i$ in component C_i. For an assignment $v_1 = v_2$, we use the following rule **R2**:

R2 for $v_1 = v_2 : f(G) = G \cup \{\, (v_1^i, o)_x \mid (v_2^i, o)_x \in G \,\}$.

The kind x of the new edge is the same as the kind of the old one ($x \in \{L, R\}$). The sources of the point-to edges are the component-specific copies v_1^i and v_2^i of v_1 and v_2.

The next two rules are related to reading and writing of object fields fld.

R3 for $v_1 = v_2.fld$:
$$f(G) = G \cup \{(v_1^i, o_2)_x \mid (v_2^i, o_1)_L \in G \wedge (o_1, fld, o_2)_x \in G\}.$$

R4 for $v_1.fld = v_2$:
$$f(G) = G \cup \{(o_1, fld, o_2)_x \mid (v_1^i, o_1)_L \in G \wedge (v_2^i, o_2)_x \in G\}.$$

In reading and writing of object fields, only local points-to edges are considered because fields of remote objects are not accessible through remote references. To illustrate this rule, consider the two assignments $\texttt{arr} = \texttt{new Listener}[10]$ and $\texttt{this.all} = \texttt{arr}$ on lines 15-16 in Fig. 2. After the analysis infers the local points-to edges from arr^1 to s^1_{array}

2. Actually, the initialization of $Reach_i$ should also include all library methods that are executed at JVM startup. Furthermore, during the analysis $Reach_i$ should be updated with static initializers, finalizers, and `run` methods of threads. Our implementation handles these issues.

and from $\mathtt{this}^1_{MyChannel}$ to $s^1_{MyChannel}$, rule **R4** can be used to create a points-to edge $(s^1_{MyChannel}, \mathtt{all}, s^1_{array})_L$, as shown in Fig. 3.

In the following two rules, $X.fld$ represents a static field fld declared in class X.

$$\mathbf{R5} \text{ for } v = X.fld:$$
$$f(G) = G \cup \{ (v^i, o)_x \mid (X.fld^i, o)_x \in G \}.$$
$$\mathbf{R6} \text{ for } X.fld = v:$$
$$f(G) = G \cup \{ (X.fld^i, o)_x \mid (v^i, o)_x \in G \}.$$

Static fields are treated similarly to local variables. Hence, these rules are essentially the same as the rules for $v_1 = v_2$, and they use the component-specific copy $X.fld^i$ of static field $X.fld$.

4.4.3 Object Creation

The rule for processing object-creation statements in reachable code in component C_i is as follows:

$$\mathbf{R7} \text{ for } v = new\ X : f(G) = G \cup \{ (v^i, s^i)_L \}.$$

Here, $s \in S$ is the allocation site corresponding to the `new` expression. Even if the newly created object is remote (i.e., an instance of a class that implements `Remote`), the reference to it is an ordinary local reference.

4.4.4 Calls to Static Methods

Consider a call to a static method m declared in class X; suppose the call occurs in some reachable method in component C_i.

$$\mathbf{R8} \text{ for } w = X.m(v_1, \ldots, v_k) : f(G) = G \cup$$
$$\{ (p^i_t, o)_x \mid (v^i_t, o)_x \in G \wedge 1 \le t \le k \} \cup$$
$$\{ (w^i, o)_x \mid (ret^i, o)_x \in G \}.$$

In this rule, p_t are the formal parameters of the called static method $X.m$. We use ret to denote a special artificial local variable in $X.m$ that is assigned all and only return values of the method; static analyses often introduce such helper variables for convenience. The effects of parameter passing and return values are essentially the same as in rule **R2**.

4.4.5 Calls to Instance Methods

Consider an instance call $w = v_0.m(v_1, \ldots, v_k)$ occurring in some reachable method in component C_i.

$$\mathbf{R9} \text{ for } w = v_0.m(v_1, \ldots, v_k) : f(G) = G \cup$$
$$\{ ResolveLocal(G, m, o, v^i_1, \ldots, v^i_k, w^i) \mid (v^i_0, o)_L \in G \} \cup$$
$$\{ ResolveRemote(G, m, s^j, v^i_1, \ldots, v^i_k, w^i) \mid (v^i_0, s^j)_R \in G \}.$$

Case 1: Local Calls. For calls made through local references, we have

$$\begin{aligned}
&ResolveLocal(G, m, o, v^i_1, \ldots, v^i_k, w^i) = \\
&\quad \text{let } m'(p_0, p_1, \ldots, p_k, ret) \text{ be the result of } dispatch(o, m) \\
&\quad \text{add } m' \text{ to } Reach_i \\
&\quad \text{return } \{ (p^i_0, o)_L \} \cup \\
&\qquad \{ (p^i_t, o')_x \mid (v^i_t, o')_x \in G \wedge 1 \le t \le k \} \cup \\
&\qquad \{ (w^i, o')_x \mid (ret^i, o')_x \in G \}.
\end{aligned}$$

The runtime target method m' is determined based on the type of o and on the compile-time target m, using helper function *dispatch*, which encodes the rules for runtime virtual dispatch. The implicit formal parameter `this` in m' is represented by p_0, and the explicit formal parameters are $p_1, \ldots, p_k$. The propagation of values from actual parameters v^i_t to formal parameters p^i_t is the same as in rule **R8** for static calls. Similarly, the modeling of the return value of the called method is the same as in **R8**.

Case 2: Remote Calls. For remote invocations from component C_i to a remote object s^j in component C_j, we have

$$\begin{aligned}
&ResolveRemote(G, m, s^j, v^i_1, \ldots, v^i_k, w^i) = \\
&\quad \text{let } m'(p_0, p_1, \ldots, p_k, ret) \text{ be the result of } dispatch(s^j, m) \\
&\quad \text{add } m' \text{ to } Reach_j \\
&\quad \text{return } \{ (p^j_0, s^j)_L \} \cup \\
&\qquad \{ (p^j_t, o')_R \mid (v^i_t, o')_x \in G \wedge 1 \le t \le k \wedge o' \text{ is remote} \} \\
&\qquad \cup \{ (w^i, o')_R \mid (ret^j, o')_x \in G \wedge o' \text{ is remote} \} \cup \\
&\qquad ResolveSerialization(G, v^i_1, \ldots, v^i_k, p^j_1, \ldots, p^j_k) \cup \\
&\qquad ResolveSerialization(G, ret^j, w^i).
\end{aligned}$$

To illustrate this part of rule **R9**, consider the call `f.add(g)` on line 30 in Fig. 2. Due to the remote points-to edge from $\mathtt{f}^2$ to $s^1_{MyChannel}$ (created by rule **R1**), the analysis needs to compute

$$ResolveRemote(G, \mathtt{Channel.add}, s^1_{MyChannel}, \mathtt{g}^2).$$

Here, `Channel.add` is the compile-time target method of the call, because the compile-time type of `f` is `Channel`. The runtime target method for receiver object $s^1_{MyChannel}$ is method `MyChannel.add` in component C_1. Suppose that G contains an edge $(\mathrm{g}^2, s^2_{MyListener_1})_L$, due to

$$\mathtt{g} = \mathtt{new\ MyListener}()$$

on line 30. In this case, *ResolveRemote* will add to G an edge $(\mathtt{this}^1_{add}, s^1_{MyChannel})_L$ representing the receiver object of the remote call, as well as an edge $(\mathrm{c}^1, s^2_{MyListener_1})_R$ representing the flow of values from actual parameter g^2 to formal parameter c^1.

In the general case, the invoked remote method m' in component C_j is determined based on the same rules for virtual dispatch that are used for ordinary nonremote calls [26]. The invocation creates a local points-to edge from `this` in m' to the remote object s^j. For actual parameters v^i_t that point to remote objects o', remote references to o' are created for the corresponding formal parameters p^j_t of m'. Note that edge (v^i_t, o') could be either local or remote. If the return value of m' is a (local or remote) reference to a remote object o', the left-hand-side variable w^i at the call site starts pointing remotely to o'.

Functions *ResolveLocal* and *ResolveRemote* can be easily augmented to construct the call (multi)graph of the application. The nodes in the call graph are pairs (m, i), where method m belongs to $Reach_i$. The edges correspond to call statements: If statement st in method m in component C_i invokes method n in component C_j ($i = j$ or $i \neq j$), the call graph contains an edge from (m, i) to (n, j),

labeled with st. Our implementation builds the call graph on the fly, during the analysis.

4.5 Modeling of Nonremote Actual Parameters

Function *ResolveSerialization* models parameter passing for nonremote actual parameters. Recall that for each object name s^i which represents nonremote serializable runtime objects created by allocation site s in component C_i, the analysis defines a set of object names $s^{i,j}$ for copy objects, one for each component C_j. For convenience, for each component C_j, we define the following map μ_j:

- $\mu_j(s^i) = s^{i,j}$ when s^i is a nonremote serializable object created in some C_i.
- $\mu_j(s^{k,i}) = s^{k,j}$ when $s^{k,i}$ is an object created in some C_i as a deserialized copy of ordinary object s^k.
- $\mu_j(s^i) = s^i$, when s^i is a remote object in some C_i.

Given an object name o that represents runtime objects in some component C_i, object name $\mu_j(o)$ represents the corresponding runtime objects in C_j. For example, in Fig. 3, for the nonremote serializable object s^3_{Event}, we have $s^{3,1}_{Event} = \mu_1(s^3_{Event})$, and $s^{3,2}_{Event} = \mu_2(s^{3,1}_{Event})$.

The effects of a remote call $v_0.m(v_1, \ldots, v_k)$ on nonremote parameters are as follows: The object graph reachable from $v_1, \ldots, v_k$ is traversed according to the rules described below. All traversed nonremote serializable objects are serialized and recreated in the target component. This process can be described by defining a subgraph *Copied*:

- If $(v^i_t, o)_L \in G$ and o is a nonremote serializable object, then $(v^i_t, o)_L \in Copied$.
- If $o \in Copied \wedge (o, fld, o')_L \in G$, where fld is a nontransient field and o and o' are nonremote serializable objects, then $(o, fld, o')_L \in Copied$.
- If $o \in Copied \wedge (o, fld, o')_x \in G$, where fld is a nontransient field, o is a nonremote serializable object, and o' is a remote object, then $(o, fld, o')_x \in Copied$.
- $Copied$ is the smallest set with these properties.

If a field is declared as transient, its value is not subjected to further serialization. If a nontransient field points to a remote object (either locally or remotely; $x \in \{L, R\}$), the traversal stops and the remote object is not serialized. However, if the field points to a nonremote object, serialization is attempted; if the pointed-to object is not serializable, an exception is thrown. The definition of $Copied$ leads to

$$
\begin{aligned}
&ResolveSerialization(G, v^i_1, \ldots, v^i_k, p^j_1, \ldots, p^j_k) = \\
&\{ (p^j_t, \mu_j(o))_L \mid (v^i_t, o)_L \in Copied \wedge o \text{ is n.r.s.} \} \cup \\
&\{ (\mu_j(o), fld, \mu_j(o'))_L \mid \\
&\quad (o, fld, o')_L \in Copied \wedge o, o' \text{are n.r.s.} \} \cup \\
&\{ (\mu_j(o), fld, \mu_j(o'))_R \mid \\
&\quad (o, fld, o')_x \in Copied \wedge o \text{ is n.r.s.} \wedge o' \text{ is remote} \}.
\end{aligned}
$$

Here, "n.r.s." stands for "nonremote but serializable." The serialization mechanism initializes a copy object (i.e., a deserialized object) not by invoking a constructor of its class but, rather, by invoking the no-arguments constructor of the "lowest" nonserializable superclass. It is easy to add this invocation to the rules and, for simplicity, we omit this detail from the presentation.

5 Analysis Algorithm

This section describes an algorithm for implementing the points-to analysis whose theoretical foundations were presented in the previous section. Rules **R1** through **R9**, introduced in the previous section, model the effects of each type of Java statements. The algorithm described in this section implements the semantics of these rules.

Our approach is based on techniques proposed by Lhoták and Hendren [21] for analysis of nondistributed Java applications. We define several extensions and generalizations of their approach in order to enable analysis of distributed programs.

5.1 Pointer Assignment Graph

The analysis algorithm uses a data structure referred to as a *pointer assignment graph* (PAG). Nodes in this graph represent memory locations or expressions that refer to such locations. For a name $v^i \in V$, there is a PAG node $node(v^i)$ corresponding to this name. There are also PAG nodes of the form $node(v^i.fld)$ for each instance field fld accessed through v^i. Similarly, for each object name $o \in O$, there are PAG nodes $node(o)$ and $node(o.fld)$.

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 $Reach_i$, the PAG contains an edge $node(v^i_2) \longrightarrow node(v^i_1)$. Whenever the analysis adds a method to $Reach_i$, the statements in the body of that method are processed and the corresponding PAG edges are created. Additional details about PAG generation and use are discussed later in this section.

5.2 Algorithm Overview

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

- A pointer assignment graph (PAG).
- Local points-to set Pt_L and remote points-to set Pt_R for PAG nodes $node(v^i)$ and $node(o.fld)$.
- A set of reachable methods $Reach = \bigcup_i Reach_i$, where $m^i \in Reach_i$ represents the copy of method m in component C_i.
- A call graph with nodes $m^i \in Reach$ and edges

  $$e \in Reach \times Reach \times CallSites \times \{L, R\}.$$

  A remote call graph edge $(m^i, n^j, c)_R$ 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 $(m^i, n^i, c)_L$ 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 [20], [21] for non-RMI Java—we introduce new techniques in order to handle remote references, remote calls, and serialization. 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

input program code
output $PAG = (N, E)$; initialized to $(\emptyset, \emptyset)$
$Pt_L : (V \cup (O \times F)) \rightarrow 2^O$; $Pt_R : (V \cup (O \times F)) \rightarrow 2^O$
for all x, $Pt_{L/R}(x)$ are initialized to $\emptyset$
$Reach$: set of reachable methods; initialized to $\emptyset$
$CallGraph$: call graph; the set of nodes is $Reach$;
the call graph is initialized to $(\emptyset, \emptyset)$
declare $NodeWorklist$: set of PAG nodes; initialized to $\emptyset$
$MethodWorklist$: set of methods; initialized to $\emptyset$
[1] $Reach := \bigcup_i \; \{MainMethod^i\} \cup \{StaticInitializerForStartClass^i\} \cup \{JVMStartupMethods^i\}$
[2] **foreach** $m^i \in Reach$ **do** $ProcessBody(m^i)$
[3] $PropagatePointsToSets$

Fig. 4. Top level of the analysis algorithm.

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.

5.3 Top Level of the Algorithm

The top-level functionality of the algorithm is shown in Fig. 4. For each component C_i, line 1 initializes *Reach* with 1) the main method of the component, 2) the static initializer (if any) of the class containing that main method, and 3) the set of library methods that are executed in C_i at JVM startup, before the main method is invoked. We assume that all initialization code for static fields inside a class (i.e., initialization expressions in field declarations [12, Section 8.3.2.1] and static initialization blocks [12, Section 8.7]) is combined in an artificial static method *StaticInitializer* for that class. Since the class containing the main method for a component C_i could be initialized as a result of the invocation of that main method, the code in *StaticInitializer* should be considered executable and this artificial method should be added to *Reach*.

5.4 Processing a Newly Discovered Reachable Method

For each method m^i in the initial set *Reach*, the analysis processes the method body using procedure *ProcessBody* (line 2 in Fig. 4). Later, during the rest of the analysis, this procedure is also executed (exactly once) on each newly discovered reachable method. PAG edges are created to represent the value-flow semantics of the statements inside the method body (lines 4-13 in Fig. 5), excluding the effects of method calls. For an assignment $v = new\ X$, based on rule **R7** from Section 4.4, the analysis adds s^i to $Pt_L(v^i)$ and puts $node(v^i)$ on a worklist *NodeWorklist* of PAG nodes. Throughout the algorithm, the elements of this worklist are

procedure $ProcessBody(m^i)$
[4] **foreach** statement st in m^i **do**
[5] **if** st is $v = new\ X$ **then**
[6] add edge $node(s^i) \longrightarrow node(v^i)$ to PAG
[7] $Pt_L(v^i) := Pt_L(v^i) \cup \{s^i\}$
[8] add $node(v^i)$ to $NodeWorklist$
[9] **if** st is $v_1 = v_2$ **then**
add edge $node(v_2^i) \longrightarrow node(v_1^i)$ to PAG
[10] **if** st is $v_1 = v_2.fld$ **then**
add edge $node(v_2^i.fld) \longrightarrow node(v_1^i)$ to PAG
[11] **if** st is $v_1.fld = v_2$ **then**
add edge $node(v_2^i) \longrightarrow node(v_1^i.fld)$ to PAG
[12] **if** st is $v = X.fld$ **then**
add edge $node(X.fld^i) \longrightarrow node(v^i)$ to PAG
[13] **if** st is $X.fld = v$ **then**
add edge $node(v^i) \longrightarrow node(X.fld^i)$ to PAG
[14] **foreach** method $n^j \in Reach$ for which $ProcessBody$ has already been executed **do**
[15] $ProcessInitialRemoteReferences(m^i, n^j)$
[16] **foreach** monomorphic call site c in m^i where the invoked method is n^i **do**
[17] $AddCallGraphEdge(m^i, n^i, c, L)$

Fig. 5. Processing the body of a newly discovered reachable method.

PAG nodes $node(v^i)$ whose points-to sets may need to be propagated to other PAG nodes.

Next, we consider all pairs of local variables that could be used to "bootstrap" initial remote references between m^i and reachable methods n^j (lines 14-15). For brevity, *ProcessInitialRemoteReferences* is not shown. This helper procedure considers all pairs of local variables in $I_{i \rightarrow j}$ and $I_{j \rightarrow i}$ such that one variable is local to m^i and the other one is local to n^j. The analysis creates special PAG edges between these locals; these edges are labeled "remote." Recall that for the code in Fig. 2, due to lines 23, 29, and 35, we have $I_{1 \rightarrow 2} = \{(\mathtt{e}^1, \mathtt{f}^2)\}$ and $I_{1 \rightarrow 3} = \{(\mathtt{e}^1, \mathtt{h}^3)\}$. During the processing of the three main methods, the algorithm will create PAG edges $node(\mathtt{e}^1) \stackrel{remote}{\longrightarrow} node(\mathtt{f}^2)$ and $node(\mathtt{e}^1) \stackrel{remote}{\longrightarrow} node(\mathtt{h}^3)$ to represent the intercomponent flow of reference values due to the naming service. Fig. 7, which contains a subset of the PAG for the running example, shows these two edges. 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^i, the corresponding call graph edges are created using procedure *AddCallGraphEdge* (this procedure is discussed below). This is done for each call site c that

procedure $AddCallGraphEdge(m^i, n^j, c, x)$

[18] **if** $(m^i, n^j, c)_x \in CallGraph$ **then** return

[19] add $(m^i, n^j, c)_x$ to $CallGraph$

[20] **if** $n^j \notin Reach$ **then**

  add n^j to $Reach$ and to $MethodWorklist$

[21] **foreach** actual parameter v_t^i of c with corresponding formal parameter p_t^j of n^j **do**

[22] add edge $node(v_t^i) \xrightarrow{x} node(p_t^j)$ to PAG

[23] add $node(v_t^i)$ to $NodeWorklist$

[24] add edge $node(ret^j) \xrightarrow{x} node(w^i)$ to PAG, where the return value at c is assigned to w

[25] add $node(ret^j)$ to $NodeWorklist$

Fig. 6. Creating a new call graph edge and the corresponding PAG edges.

represents a static call, a constructor call, or a nonpolymorphic instance call (e.g., call to a final method). Note that remote calls are never processed in this step, because such calls are always performed through remote interfaces. Polymorphic calls (including remote calls) are processed later, when the points-to sets of PAG nodes can be used to infer that certain receiver objects are possible at these call sites.

5.5 Adding a Call Graph Edge

Procedure *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. The actual parameters are the calling method m^i, the target method n^j, the call site c, and a value $x \in \{L, R\}$ signifying a local or a remote call. If edge $(m^i, n^j, c)_x$ is new, it is added to the call graph. If the target method n^j is discovered for the first time, it is added to the set of reachable methods. The new method is also added to a worklist *MethodWorklist* of newly discovered reachable methods; as discussed later, each method on this worklist is eventually processed using *ProcessBody*. Each reachable method (except for the initial methods from line 1 in Fig. 4) is added to *MethodWorklist* exactly once.

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* because their points-to sets may contain objects that need to be propagated to the PAG nodes for p_t^j and w^i.

As an example, consider the remote calls to `add` on lines 30 and 31 in Fig. 2. Because `g` is used as an actual parameter of these calls, the analysis creates a remote PAG edge from g^2 to the corresponding formal parameter c^1, as shown in Fig. 7. This edge represents the intercomponent flow of reference values due to parameter passing at the remote call.

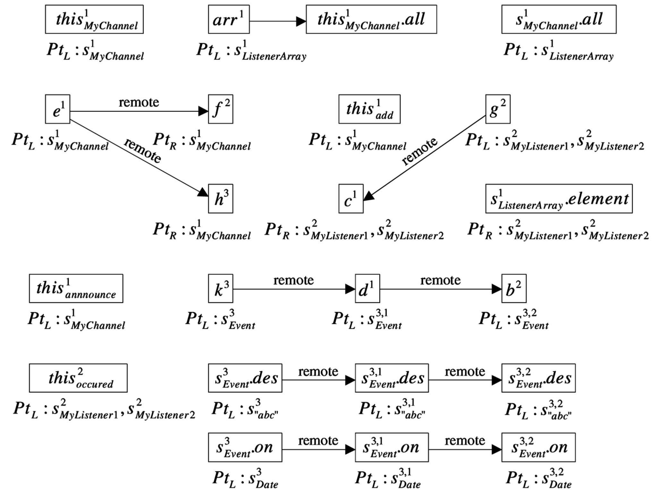

Fig. 7. Sample PAG nodes, edges, and points-to sets.

5.6 Propagation of Points-To Sets

Procedure *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. The completion is signaled by two conditions (line 41):

1. Worklist *NodeWorklist* is empty, meaning that there are no more variables with points-to sets that need to be propagated.
2. Boolean flag *SerializableToPropagate* is *false*, meaning that there are no more object fields $o.fld$ whose points-to sets need to be propagated. As described below, a value *true* for this flag indicates that points-to sets may have changed for object fields that are subject to serialization at remote calls and, therefore, further propagation may be necessary.

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

First, the analysis considers the current values of $Pt_{L/R}(v^i)$ and attempts to grow the call graph using helper procedure *BuildCallGraphOnTheFly* (line 29). For brevity, this helper procedure is not shown. Inside the procedure, all new call graph edges related to v^i—due to calls $v.m(\ldots)$ in component C_i—are created by calling *AddCallGraphEdge*. Furthermore, the points-to sets of the corresponding formal parameters `this` are updated. Finally, this helper procedure considers all the methods currently in *MethodWorklist*, as well as all newly discovered reachable methods, and processes them with *ProcessBody*. At the end of *BuildCallGraphOnTheFly*, *MethodWorklist* is empty, which guarantees that the bodies of all reachable methods have been processed.

After all necessary changes to the call graph have occurred, the PAG edges related to v^i are examined and the corresponding points-to sets are updated (lines 30-33). The processing of these edges is a generalization of the points-to analysis algorithm from [20], [21]. The helper

```
procedure PropagatePointsToSets
declare   RemoteFieldEdges : set of PAG edges
          SerializableToPropagate : boolean
[26] repeat
[27]   repeat
[28]     remove node(v^i) from NodeWorklist
[29]     BuildCallGraphOnTheFly(v^i)
[30]     foreach node(v^i) ⟶ node(w^i) in PAG do
                 ProcessSimpleEdge(node(v^i), node(w^i))
[31]     foreach node(v^i) ⟶ node(w^i.fld) in PAG do
                 ProcessStoreEdge(node(v^i), node(w^i.fld))
[32]     foreach node(w^i) ⟶ node(v^i.fld) in PAG do
                 ProcessStoreEdge(node(w^i), node(v^i.fld))
[33]     foreach node(v^i.fld) ⟶ node(w^i) in PAG do
                 ProcessLoadEdge(node(v^i.fld), node(w^i))
[34]     foreach node(v^i) --remote--> node(w^j) in PAG do
                 ProcessRemoteEdge(node(v^i), node(w^j))
[35]   until NodeWorklist = ∅
[36]   foreach node(v^i) ⟶ node(w^i.fld) in PAG do
               ProcessStoreEdge(node(v^i), node(w^i.fld))
[37]   foreach node(v^i.fld) ⟶ node(w^i) in PAG do
               ProcessLoadEdge(node(v^i.fld), node(w^i))
[38]   RemoteFieldEdges :=
         { e ∈ PAG | e is edge node(o_1.fld) --remote--> node(o_2.fld) }
[39]   SerializableToPropagate := false
[40]   foreach e ∈ RemoteFieldEdges do
               ProcessRemoteFldEdge(source(e), target(e))
[41] until NodeWorklist = ∅ ∧ SerializableToPropagate = false
```

Fig. 8. Propagation of objects to points-to sets.

procedures *ProcessSimpleEdge*, *ProcessLoadEdge*, and *ProcessStoreEdge* correspond to rules **R2**, **R3**, and **R4** from Section 4.4, respectively. In each of these procedures, if the local or remote points-to set of a PAG node $node(v^i)$ grows, the node is added to the node worklist for further propagation. These procedures are not discussed further in this paper. For illustration, consider assignments $\mathrm{arr} = \mathtt{new\ Listener[10]}$ and $\mathrm{this.all} = \mathtt{arr}$ on lines 15-16 in Fig. 2. As shown in Fig. 7, due to the effect of the first assignment (captured by rule **R7**), the local points-to set of $\mathtt{arr}^1$ contains $s^1_{ListenerArray}$, and the local points-to set of $\mathrm{this}^1_{MyChannel}$ contains $s^1_{MyChannel}$. Due to local PAG edge

$$node(\mathtt{arr}^1) \longrightarrow node(\mathtt{this}^1_{MyChannel}.\mathtt{all}),$$

procedure *ProcessStoreEdge* will eventually add $s^1_{ListenerArray}$ to the local points-to set of the PAG node for $s^1_{MyChannel}.\mathtt{all}$.

At line 34 in Fig. 8, the algorithm performs propagation along PAG edges that are labeled as remote. Recall that such edges could be created due to the effects of the naming service (line 15 in Fig. 5) or parameter passing at remote calls (lines 22 and 24 in Fig. 6). Helper procedure

```
procedure ProcessRemoteEdge(node(v^i), node(w^j))
[42] Pt_R(w^j) := Pt_R(w^j) ∪ { o | o ∈ Pt_L(v^i) ∧ o is remote }
[43] Pt_R(w^j) := Pt_R(w^j) ∪ Pt_R(v^i)
[44] foreach non-remote serializable o ∈ Pt_L(v^i) do
[45]   Pt_L(w^j) := Pt_L(w^j) ∪ { μ_j(o) }
[46]   foreach non-transient field fld of o do
[47]     add edge node(o.fld) --remote--> node(μ_j(o).fld) to PAG,
         if the edge does not exist already
[48] if Pt_L(w^j) or Pt_R(w^j) changed then
       add node(w^j) to NodeWorklist

procedure ProcessRemoteFldEdge(node(o.fld), node(μ_j(o).fld))
[49] Pt_R(μ_j(o).fld) := Pt_R(μ_j(o).fld) ∪
       { o' | o' ∈ Pt_L(o.fld) ∧ o' is remote }
[50] Pt_R(μ_j(o).fld) := Pt_R(μ_j(o).fld) ∪ Pt_R(o.fld)
[51] foreach non-remote serializable o' ∈ Pt_L(o.fld) do
[52]   Pt_L(μ_j(o).fld) := Pt_L(μ_j(o).fld) ∪ { μ_j(o') }
[53]   foreach non-transient field fld2 of o' do
[54]     add node(o'.fld2) --remote--> node(μ_j(o').fld2) to PAG,
         if the edge does not exist already
[55] if Pt_L(μ_j(o).fld) or Pt_R(μ_j(o).fld) changed then
       SerializableToPropagate := true
```

Fig. 9. Propagation along remote PAG edges.

ProcessRemoteEdge is shown in Fig. 9. This procedure propagates objects to the remote points-to sets of the target nodes of remote PAG edges (lines 42-43). This propagation implements the semantics of rules **R1** and **R9** from Section 4. For example, Fig. 7 shows that there is a remote PAG edge from $\mathtt{e}^1$ to $\mathtt{f}^2$. Since the local points-to set of $\mathtt{e}^1$ contains the remote object $s^1_{MyChannel}$, this object is added to the remote points-to set of $\mathtt{f}^2$, according to the semantics of rule **R2**.

Furthermore, the procedure adds copy objects to local points-to sets (line 45) and creates additional remote edges corresponding to serialization at parameter passing for remote calls (line 47), implementing the semantics of rule **R9**. Each such new edge is of the form $node(o.fld) \overset{remote}{\longrightarrow} node(\mu_j(o).fld)$ and represents the flow of values from field $o.fld$ to the deserialized copy of that field in component C_j. For example, in Fig. 7, there is a remote PAG edge from $\mathtt{k}^3$ to $\mathtt{d}^1$. Since the local points-to set of $\mathtt{k}^3$ contains the nonremote serializable object s^3_{Event}, the corresponding copy object $s^{3,1}_{Event} = \mu_1(s^3_{Event})$ is added to the local points-to set of $\mathtt{d}^1$. The PAG is also updated with edges

$$node(s^3_{Event}.\mathtt{des}) \overset{remote}{\longrightarrow} node(s^{3,1}_{Event}.\mathtt{des})$$

and $node(s^3_{Event}.\mathtt{on}) \overset{remote}{\longrightarrow} node(s^{3,1}_{Event}.\mathtt{on})$.

After the node worklist is processed completely by the inner loop in *PropagatePointsToSets*, the outer loop (lines 26-41) considers all existing load and store edges in the PAG

(lines 36-37) and performs additional propagation for them. As discussed in [20], this is necessary to propagate points-to relationships that may be missed by the inner loop due to aliasing.

Lines 38-40 are needed to propagate the points-to sets of fields $o.fld$ along remote PAG edges that correspond to serialization. Each such edge is of the form

$$node(o.fld) \stackrel{remote}{\longrightarrow} node(\mu_j(o).fld).$$

In procedure *ProcessRemoteFldEdge*, using the points-to sets of $o.fld$, the analysis updates the points-to sets of $\mu_j(o).fld$ (lines 49-50 and 52 in Fig. 9). This process may also add new remote edges between object fields (lines 54). If the points-to sets of $\mu_j(o).fld$ change, flag *SerializableToPropagate* is raised to ensure that at least one more iteration of the outer loop in *PropagatePointsToSets* is executed, in order to propagate these changed sets. It is easy to show that when new remote PAG edges are created at line 54, line 55 is guaranteed to raise the flag, thus forcing future propagation along those new edges. When the flag remains *false* after line 40, neither the PAG nor the points-to sets have changed due to the effects of object serialization. Together with $NodeWorklist = \emptyset$, this guarantees that further propagation is not necessary.

6 Handling of the Standard Java Libraries

The standard Java libraries are implicitly added to the set of classes $cls(C_i)$ for each component. Based on the analysis definition presented earlier, library variables and objects will have multiple copies. For example, if a library method m has a local variable v, the points-to analysis will use multiple copies of v—that is, a separate name v^i for each component C_i. Object names are treated similarly. Our experiments with this approach showed that the majority of analysis time is spent on processing the relevant code from the libraries. Even when the size of the nonlibrary code is small, the necessary conservative treatment of various features from the libraries (e.g., JVM startup, initialization of static fields, dynamic class loading and reflection, finalizers, etc.) requires the analysis to consider a large number of library methods as reachable. The replication of library variables and objects results in significant running time for the analysis.

To reduce running time, we designed and implemented an alternative technique for handling the standard libraries. The basic idea is to create only one replica of a library entity. For a variable v, we use a single name v^{lib} instead of multiple names v^i. For an object allocation site s, there is a single object name s^{lib}. The analysis also maintains a set of reachable methods $Reach_{lib}$, and library methods are added to this set rather than to the component sets $Reach_i$.

The rules for PAG construction can be modified in a corresponding manner. For example, if an assignment $v_1 = v_2$ is in the body of a reachable library method $m \in Reach_{lib}$, the analysis creates PAG edge $node(v_2^{lib}) \longrightarrow node(v_1^{lib})$. As another example, consider an assignment $v = X.fld$ from a static field to a local variable v. If the assignment appears in a reachable library method, edge $node(X.fld^{lib}) \longrightarrow node(v^{lib})$ should be created. On the other hand, if the assignment is discovered in some reachable nonlibrary method $m^i \in Reach_i$, one of the following edges should be created: $node(X.fld^{lib}) \longrightarrow node(v^i)$ if X is a library class, or $node(X.fld^i) \longrightarrow node(v^i)$ otherwise. As another example, whenever a nonlibrary method calls a library method, the actual-to-formal PAG edges are of the form $node(v^i) \longrightarrow node(p^{lib})$. Likewise, edges

$$node(ret^{lib}) \longrightarrow node(w^i)$$

are created to represent the flow of return values. Similar treatment is necessary for a callback from a library method to a nonlibrary one.

The propagation of points-to sets along PAG edges follows the algorithm outlined in Section 5. However, it is possible to filter out some of the objects that are being propagated, in cases when the component labels do not match. For example, consider some $o \in Pt_L(v^{lib})$ and an edge $node(v^{lib}) \longrightarrow node(w^i)$. If o is a nonlibrary object s^j, it is propagated to the points-to set of w^i only if $i = j$. More generally, filtering can be used to ensure that the local points-to set of any v^i or $s^i.fld$ does not contain any nonlibrary objects s_2^j for which $i \neq j$. Note that this filtering cannot be applied to remote points-to sets, because the elements of these sets could be nonlibrary objects created in arbitrary components.

After the completion of the analysis, the local points-to sets for non-library variables and objects are processed to replace names s^{lib}. For example, if $Pt_L(v^i)$ contains s^{lib}, this object name can stand only for objects created in component C_i; thus, s^{lib} can be replaced by s^i. Note that such a replacement cannot be performed for $Pt_R(v^i)$ because, in this case, s^{lib} represents objects in any component and not necessarily objects in C_i.

The full-replication approach from Section 4 and the zero-replication approach from above are the two endpoints of the design spectrum for handling of the standard libraries. Since the degree of replication has a direct effect on both analysis cost and analysis precision, future investigations should be performed in order to understand thoroughly this entire spectrum of cost-precision trade-offs.

7 Analyses for Program Understanding

Points-to information is a frequently required "enabler" for a wide range of other techniques. This section briefly discusses three specific uses of the points-to analysis for the purposes of program understanding of RMI-based applications. Of course, many other uses are possible (e.g., for program slicing, change impact analysis, etc.).

7.1 Call Graph

As discussed Section 4.4, the analysis performs on-the-fly call graph construction. The resulting graph can serve as the starting point for many other static analyses. The call graph can also be used to answer questions such as "Given a call statement st in component C_i, which methods in other components may be invoked by st, directly or transitively?" This and similar questions can enhance the understanding of the intermethod and intercomponent flow of control, especially when combined with GUI-based browsing tools that display graphically the relevant parts of the call graph.

7.2 Data Dependencies

Consider a component C_i and some object s^i created in this component. A statement st_1^i in C_i could potentially read or write some field of s^i (either directly or transitively through its callees). Now consider a call site st_2^j in some other component C_j, and suppose st_2^j invokes some remote method from C_i. Due to the remote call, the execution of st_2^j could (transitively) read or write some field of object s^i. Thus, it is possible to have a read-write or write-read dependence between st_1^i and st_2^j. The pair (st_1^i, st_2^j) represents a potential *intercomponent data dependence* between C_i and the caller component C_j. Furthermore, consider another call site st_3^k in a third component C_k, and suppose st_3^k invokes some remote method from C_i. It is possible to have a dependence between st_2^j and st_3^k due to some field of s^i. In this case, the intercomponent dependence is between C_j and C_k, but the memory responsible for the dependence is in the JVM for C_i.

We have defined and implemented an algorithm that, for a given component C_i, computes all pairs (st_1^i, st_2^j) and (st_2^j, st_3^k) that correspond to potential data dependencies, as defined above. To illustrate the algorithm, consider component C_1 from Fig. 1. The call `h.announce(k)` in `main` in C_3 creates a copy object $s_{Event}^{3,1}$ in C_1 (based on s_{Event}^3 in C_3) and initializes its fields on and des with copy objects $s_{Date}^{3,1}$ and $s_{"abc"}^{3,1}$. Consider now `all[i].occurred(d)` in `announce` in C_1. Since the local points-to set of actual parameter d^1 contains $s_{Event}^{3,1}$, we can determine that, due to the serialization, the values of fields $s_{Event}^{3,1}.on$ and $s_{Event}^{3,1}.des$ are read. Thus, there is a write-read dependence between the call to `announce` in C_3 and the call to `occurred` in C_1, due to memory locations $s_{Event}^{3,1}.on$ and $s_{Event}^{3,1}.des$. As another example, consider `f.add(g)` in `main` in C_2 and `h.announce(k)` in `main` in C_3. The call to `add` results in a modification of $s_{MyChannel}^1.num$, due to `num++`. Since `announce` reads the value of $s_{MyChannel}^1.num$, there is a dependence between `f.add(g)` in C_2 and `h.announce(k)` in C_3.

The computation of such dependencies starts by examining the local points-to set at reads and writes of expressions $v.fld$. For each statement $st^i \in Reach_i$ of the form $v_1.fld = v_2$, the analysis defines a set

$$Mod(st^i) = \{o.fld \,|\, o \in Pt_L(v_1^i)\}.$$

Similarly, for $v_1 = v_2.fld$ we have

$$Use(st^i) = \{o.fld \,|\, o \in Pt_L(v_2^i)\}.$$

The reads and writes of static fields are processed in a similar fashion. The analysis also needs to take into account the reads and writes performed during object serialization and deserialization. For an instance call $w = v_0.m(v_1, \ldots, v_k)$ where $Pt_R(v_0^i) \neq \emptyset$, set $Use(st^i)$ should include

$$\{o.fld \,|\, (o, fld, o')_x \in Copied \wedge x \in \{L, R\}\},$$

where $Copied$ is defined in Section 4.5. In other words, any object field that is examined during the serialization process should be part of the Use set of the corresponding call. Note that we include both $o.fld$ that point to nonremote objects and $o.fld$ that point to remote objects. For the latter, even though serialization is not applied (the remote object in not serialized), the object field is still examined by the serialization mechanism. In addition, the remote call initializes the deserialized objects in the callee component C_j; therefore, $Mod(st^i)$ should include $\{\mu_j(o).fld \,|\, o.fld \in Use(st^i)\}$.

After the initial Mod and Use sets are computed as described above, the analysis performs iterative backward propagation of this information along the call graph edges, from callees to callers. This propagation computes the transitive read/write relationships due to method calls. Given the final solution, the intersections of sets Mod and Use for pairs of statements can be used to identify potential data dependencies.

7.3 Customized Serialization

One of the performance bottlenecks for RMI is the serialization and deserialization of nonremote actual parameters [25], [31]. Several optimizations can be used to reduce this cost. For example, if the types of the serialized objects are unique and known in advance, specialized serialization code can be created rather than using the more expensive default serialization mechanism. As another example, if the object graph that will be serialized is always acyclic, a cheaper version of the serialization algorithm can be used, as opposed to the general version which must detect cycles. Such techniques have been shown to be quite effective in reducing the cost of serialization in RMI applications [50]. By analyzing the structure of the points-to graph produced by our analysis, it is straightforward to expose these optimization opportunities to a programmer. This information enables the introduction of customized serialization, either manually (through methods `writeObject` and `readObject` [27]), or automatically with the help of an optimization tool.

7.4 Other Potential Uses

Testing of distributed Java applications can be based on adequacy criteria that consider the coverage of start-to-end scenarios [4]; the corresponding execution paths can be automatically constructed (and monitored at runtime) based on the call graph. As another example, the call graph and the data dependencies may be useful for static analyses that attempt to identify potential deadlocks and race conditions in RMI-based Java software.

8 Experimental Study

We implemented the points-to analysis algorithm using the Soot framework [49], version 2.1, and the Spark component of Soot which implements the points-to analysis techniques from [21] for nondistributed Java programs. The analysis in Spark uses state-of-the-art analysis techniques and provides the basis for our own implementation, including the handling of issues such as JVM startup, native methods, reflection, etc. The analysis was executed on a 2.8 GHz Pentium 4 PC with 3 GB of memory, using Sun's HotSpot Client JVM 1.4.2 for Windows, with maximum JVM heap size 1.5 GB (option `Xmx`).[3] The experiments were performed on the set of RMI-based Java applications listed in Table 1.

3. Table 2 provides additional results for other memory configurations.

TABLE 1
Subject Programs, Reachable Methods, and Running Times

(1) Application	(2) # C_i	(3) Classes			(4) MethodsOrig		(5) MethodsAppr		(6) Time (sec)		
		Prog	Orig	Appr	Prog	All	Prog	All	Orig	Appr	NoDstr
filesrv	2	7	2513	1258	14	14397	14	7209	1126.3	339.0	310.3
stocks	2	8	2515	1263	20	14443	21	7291	1127.7	340.2	312.3
rmttask	2	9	2514	1261	12	14390	12	7204	1140.3	335.9	319.9
channel	3	11	3767	1261	18	21587	18	7210	2368.6	336.1	308.2
bank	2	14	2518	1265	21	14402	21	7216	1135.0	341.5	323.7
auction	2	17	2524	1262	62	14475	62	7321	1164.3	343.5	309.3
jodl	2	29	2551	1289	125	14630	125	7387	1190.6	358.1	330.9
jenut	2	33	2546	1292	68	14533	68	7341	1177.6	353.9	318.7
translator	2	38	2526	1275	85	14454	85	7299	1206.7	368.7	329.9
database	2	67	2583	1304	62	14629	62	7400	1237.3	382.2	325.0
ssl	2	67	2587	1306	62	14642	62	7405	1227.4	381.7	377.5
library	3	69	3823	1317	414	22011	414	7637	2549.1	404.6	371.8

The applications were obtained from publicly available projects and books, and represent a variety of domains.[4] For example, `auction` implements an auctioning system: clients connect to a server and place bids for items. As another example, `jodl` uses a JOb Dispatching Library to dispatch and execute tasks on different network nodes. For the applications that were GUI-based, we created and used equivalent non-GUI versions; this was done to avoid polluting the measurements of analysis running time with the time necessary to analyze the Java GUI libraries. As discussed below, the time for library analysis is the dominant factor in the points-to analysis running time; however, the library functionality is typically irrelevant to the distributed behavior of the application, which is implemented by the nonlibrary program code.

We ran two different versions of the points-to analysis. The *original version* uses the algorithm from Section 5 and creates replicated versions of library variables and objects. The *approximate version* creates nonreplicated versions for library entities (using the approximation techniques from Section 6) in order to reduce analysis running time, possibly at the expense of some loss of precision. Both versions employed an optimization technique for the propagation of exception objects. Since the points-to analysis is flow- and context-insensitive, it does not precisely track the flow of exception objects, and it directly propagates each such object to the points-to sets of all appropriate type-compatible variables and object fields. (This conservative treatment of exceptions is standard for subset-based points-to analysis for Java.) To reduce the cost of exception-related propagation, both versions of the analysis did not replicate exception objects on per-component basis, but rather used a single component-insensitive object name per exception object, in a manner similar to the approach from Section 6. It is easy to show that this optimization technique does not affect the precision of the analysis solution.

4. We want to thank Prof. Lionel Briand from Carleton University for providing the source code for `library`.

Column 2 in Table 1 shows the number of components C_i in each application. Column 3 contains three measurements related to the number of classes involved in the analysis. Column "Prog" represents the number of nonlibrary classes in the analyzed program—that is, the sum of the sizes of $cls(C_i)$, excluding library classes. Column "Orig" shows the total numbers of classes that contain at least one reachable method in the original version of the algorithm. These are the classes "touched" by the analysis, including both program classes from column "Prog" and all relevant library classes used (directly or transitively) by these program classes. (The experiments used the standard Java libraries from Sun's J2SDK 1.4.2 distribution for Windows.) Column "Appr" shows the total numbers of classes that contain at least one reachable method in the approximate version of the algorithm, similarly to the measurements in column "Orig." The total numbers of classes for these two versions provide a high-level indication of the amount of work done by these algorithms. Note that the program classes (i.e., nonlibrary classes) represent only a small percentage of the total number of classes being analyzed.

Column 4 describes the number of reachable methods processed by the original versions of the analysis. Column "Prog" shows the number of call graph nodes for nonlibrary methods—that is, methods declared in classes from "Prog" in column 3. Column "All" shows the number of call graph nodes for both nonlibrary and library methods, corresponding to classes from "Orig" in column 3. Similarly, column 5 and its two subcolumns describe the number of reachable methods for the approximate version. Not surprisingly, the size of the call graph is significantly larger for the original version because multiple component-specific call graph nodes m^i may correspond to the same library method m. On the other hand, the approximate version has a single call graph node m^{lib} for a reachable library method m. In both versions, the analysis has to examine the bodies of all

reachable nonlibrary methods and library methods, and has to create points-to relationships for the corresponding statements. Thus, there are several thousand methods that are analyzed, even though only a small portion of them are nonlibrary methods.

8.1 Analysis Cost

Column 6 in Table 1 shows the running time of the two versions of the analysis[5] in subcolumns "Orig" and "Appr." To enable additional cost comparison, we created an artificial nondistributed version of each program. In this version the RMI mechanisms were replaced manually with standard Java mechanisms—for example, remote calls were replaced with standard calls. This non-RMI version was analyzed using the standard RMI-unaware points-to analysis in Spark [21]. In column 6, subcolumn "NoDist" shows the running time of the standard points-to analysis on the artificial non-RMI version of the program.

The experimental results in column 6 in Table 1 provide some insights about the cost and scalability of the points-to analysis. Analysis cost depends mainly on two factors: The first major factor is the cost of running the underlying points-to analysis engine for nondistributed Java programs in Spark. The second main contributor is the replication factor introduced by the multiple components in an RMI application—essentially, the cost of computing per-component points-to solutions. The comparison between columns "Orig" and "Appr" shows that the special handling of library variables and objects in the approximate version significantly reduces the cost of the analysis and eliminates much of the per-component replication cost.

The comparison between columns "Appr" and "NoDist" indicates that the main bottleneck for scalability is the points-to analysis technology for non-RMI Java applications. The worst-case asymptotic complexity of subset-based points-to analysis is cubic. However, the experience with Spark and similar analyses for non-RMI Java has shown that, in practice, these approaches essentially have linear complexity. The key scalability problem for points-to analysis of non-RMI applications is the presence of large libraries. For example, for a small program written on top of the Java 1.4 libraries, the analysis typically has to add more than 1,000 library classes and more than 7,000 library methods to the program code and has to pay the full price of analyzing all of this library code. Note that this cost is independent of the specific implementation details of Spark — any whole-program analysis that processes the relevant library code will have to pay a similar price.

The numbers in columns "All" in Table 1 are indicators of the amount of work that the analysis needs to perform, since the body of each reachable method must be processed in order to create PAG edges and to populate points-to sets. Clearly, the majority of analysis time is spent on processing the relevant code from the standard libraries. This is a well-known problem that has not been solved for whole-program points-to analysis of non-RMI Java, even though some initial ideas have been proposed to address this issue (e.g., [38], [34], [36]).

5. These measurements differ from the ones in [43], due to some modifications and enhancements of our implementation.

In summary, the main obstacle to using our analysis for large-scale software systems is the current deficiencies of standard points-to analysis technology. Analysis cost scales linearly with code size, at a rate of about 1 minute per 1,000 analyzed methods. Solving the scalability problems in the presence of large libraries for standard points-to analysis will also solve them for our analysis of RMI-based Java. In particular, the overall analysis time can be reduced if the libraries are preanalyzed once and the computed information is reused every time an application is analyzed. Similar approaches have already been used for points-to analysis for C (e.g., [38]), but it remains to be seen how they can be successfully adapted to Java.

We gathered measurements of running time and heap size of the approximate version of our algorithm for two different system configurations in order to understand the effects of some system constraints on the cost of the analysis. The results are presented in the Table 2. Both configurations use the 2.8 GHz Pentium 4 machine presented in the beginning of this section. The first configuration has 3 GB of physical memory and runs the JVM with maximal heap size of 800 MB. The second configuration has 1 GB of physical memory and runs the JVM with 300 MB maximal heap size. The two configurations yield almost identical results in memory consumption and very similar running times; thus, memory requirements do not appear to be a scalability concern.

8.2 Analysis Solution

To gain some insights about the points-to analysis solution, we gathered a variety of measurements, as summarized in Table 3. First, for each local variable v in a reachable nonlibrary method m^i, we considered the sizes of $Pt_L(v^i)$ and $Pt_R(v^i)$. The average sizes of the local points-to sets are shown in subcolumns Pt_L in columns 2 and 3. Similarly, subcolumns Pt_R show the average sizes of the nonempty remote points-to sets. All of these averages exclude variables of exception types—as described above, the analysis propagates exception objects very conservatively, and the average points-to set sizes become artificially large if exception-typed variables are included in the metric. The results in columns 2 and 3 indicate that the approximate handling of the standard libraries has some impact on the precision of the analysis solution, but this impact does not appear to be particularly significant.

The rest of Table 3 contains additional measurements based on the points-to solution. Each of these measurements was obtained first with the original version of the analysis and then again with the approximate version. However, in all cases the results were the same; thus, columns 4 through 10 apply to both versions. Consider an expression $v.m(\ldots)$ in some nonlibrary method $m' \in Reach_i$. Column 4 shows the total number of such call sites for all components; if a call site occurs in multiple components, it is counted multiple times. Column 5 contains the number of remote call sites—that is, sites for which $Pt_R(v^i)$ was not empty. Most programs have multiple remote call sites, which indicates that there may be several different kinds of remote interactions between application components.

For each site from column 5, we computed the number of distinct remote methods that were potentially invoked by the site. More precisely, consider $v.m(\ldots)$ in some method $m' \in Reach_i$. For each $s^j \in Pt_R(v^i)$, let m'' be the target method for receiver s^j. For each remote call site we

TABLE 2
Time and Memory Measurements for Other System Configurations

(1) Application	(2) -Xmx800m, 3GB		(3) -Xmx300m, 1GB	
	Time (sec)	Heap Usage (MB)	Time (sec)	Heap Usage (MB)
filesrv	337.6	160.6	347.5	160.6
stocks	328.8	163.0	364.1	163.0
rmttask	336.5	160.4	347.0	160.3
channel	331.0	161.7	344.5	161.7
bank	341.5	161.0	351.2	160.9
auction	338.1	162.0	361.4	162.0
jodl	360.9	167.3	375.4	167.3
jenut	365.8	166.0	413.4	166.0
translator	364.9	160.7	380.6	160.9
database	380.9	166.2	396.4	166.3
ssl	382.0	166.8	401.0	166.8
library	391.0	170.7	406.8	170.7

TABLE 3
Analysis Precision

(1) Application	(2) Original		(3) Approx		(4) Calls	(5) Rmt	(6) RmtTarg	(7) RmtRef		(8) Serial	(9) OpType	(10) OpCycle
	Pt_L	Pt_R	Pt_L	Pt_R				Param	Ret			
filesrv	7.3	1.0	7.5	1.0	8	5	1.0	0	0	0	0	0
stocks	87.8	1.0	88.0	1.0	12	2	1.0	1	0	2	2	2
rmttask	7.1	1.0	7.2	1.0	9	2	1.0	0	0	1	1	1
channel	5.2	1.3	5.4	1.3	11	4	1.0	2	0	4	4	4
bank	21.2	1.25	21.4	1.25	15	9	1.0	0	1	2	2	2
auction	43.4	1.0	44.3	1.0	75	5	1.0	2	0	4	4	4
jodl	121.2	1.0	121.9	1.0	158	13	1.0	0	0	2	2	2
jenut	77.3	1.23	78.3	1.72	99	36	1.0	11	9	20	20	20
translator	75.1	2.0	75.7	2.0	69	1	2.0	0	0	1	1	1
database	31.9	1.0	35.4	1.0	33	8	1.0	0	4	2	2	2
ssl	31.0	1.0	34.4	1.0	33	8	1.0	0	4	2	2	2
library	66.6	1.67	73.0	2.46	900	26	1.0	0	0	26	26	26

computed the number of distinct targets m'' based on $Pt_R(v^i)$. Column 6 shows the average number of remote target methods over the call sites from column 5. For all applications except one, the analysis resolved each remote call site to a unique target method. Since 1.0 is a lower bound for this metric, these results show that the call graphs contain precise information about the targets of remote calls.

For each remote call site, we also examined the points-to solution and determined whether there is any flow of remote references due to parameter passing. Such flow may occur when there exists an actual parameter v for which $Pt_L(v^i)$ or $Pt_R(v^i)$ contains a remote object. In column 7, subcolumn "Param" shows the number of remote call sites at which remote references may be created in the callee due to actual parameters in the caller. Remote references also may flow as return values in the case when $Pt_L(ret^j)$ or $Pt_R(ret^j)$ contains a remote object; here, ret^j denotes the artificial variable that contains the return values of the called remote method. In column (7), subcolumn "Ret" contains the number of remote call sites at which remote references may be created in the caller due to the return value from the callee. The measurements indicate that it is not unusual for RMI applications to create additional remote references at remote calls, either in the callee (through parameter passing) or in the caller (through return values). Thus, any points-to analysis needs to include techniques for handling such a flow of remote references.

Any subsequent analysis (e.g., change impact analysis) must also take into account this flow, based on the output of the points-to analysis.

We also considered $Pt_L(v^i)$ for an actual parameter v at a remote call site to determine whether serialization for nonremote parameters may occur at the site. Column 8 shows the number of sites from column 5 for which serialization may occur due to actual parameters that point to nonremote serializable objects. These results indicate that RMI applications often take advantage of the ability to use serializable objects (and more generally, serializable object graphs) as parameters of remote calls. A points-to analysis cannot expect that the nonremote actual parameters at remote call sites are always of primitive types, and therefore the analysis must model in a general manner the possible effects of serialization. Our analysis handles this issue by introducing special PAG edges connecting the original object with its deserialized copy (Section 5).

The last two columns consider the remote call sites at which serialization may occur (i.e., the sites from column 8). As described in Section 7, points-to information can be used to provide a programmer with information about call sites at which the types of the serialized objects are unique and known in advance, or the object graph that will be serialized is always acyclic. Customized serialization at such call sites can improve the performance of the application. For each site from column 8, we determined whether the type-based optimization was possible; the number of optimizable sites is shown in column 9. Similarly, for each site from column 8, we determined the shape of the serialized object graph; column 10 shows the number of sites with acyclic graphs. For our subject applications, both optimizations were possible at all remote calls at which serialization is performed.

9 Conclusions and Future Work

There are several conclusions that could be drawn from the work presented in this paper. First, it is possible to naturally generalize the existing formalisms for points-to analysis to handle RMI features (Section 4) 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. Second, the PAG-based propagation algorithm from [21] can be generalized with the help of remote PAG edges (Section 5). The specialized propagation rules for such edges allow the algorithm to model 1) the propagation of remote references, which is relatively straightforward, and 2) the propagation of references to deserialized copy objects, which requires several extensions of the algorithm. A similar approach can potentially be useful for other existing points-to analysis algorithms for non-RMI Java programs.

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 nondistributed Java programs: most of the running time is spent in the standard libraries. Third, the approximate handling of the libraries described in Section 6 can be used to significantly reduce the running time without a major reduction in analysis precision. We believe that, in the current state of the art, the approximate version of the analysis is a viable choice for a relatively precise and practical points-to analysis of RMI-based Java applications.

We consider the work presented in this paper to be a first step in a long-term research agenda for establishing a body of work on static analysis for RMI-based applications. First, obvious targets for future work are various flow- or context-sensitive points-to analyses. Such analyses could be defined as extensions of the approach from Section 4, and their scalability should be investigated carefully. Second, it is necessary to define the RMI-specific generalizations of other categories of analyses such as side-effect analysis, def-use analysis, and escape analysis. Next, these analyses should be evaluated experimentally in the context of program-understanding tools (e.g., for change impact analysis) and test-coverage tools (e.g., for round-trip-scenario coverage [4]). Finally, it is essential to generalize and evaluate these static analyses for more powerful RMI-based middleware platforms such as Enterprise JavaBeans.

Acknowledgments

The authors would like to thank the *IEEE Transactions on Software Engineering* and International Conference on Software Maintenance reviewers for their thorough comments and suggestions. Their feedback was essential for improving the contents and the presentation of this paper.

References

[1] A. Ahern and N. Yoshida, "Formalising Java RMI with Explicit Code Mobility," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 403-422, 2005.

[2] L.O. Andersen, "Program Analysis and Specialization for the C Programming Language," PhD thesis, Dept. of Computer Science, Univ. of Copenhagen, 1994.

[3] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, "Points-to Analysis Using BDDs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 103-114, 2003.

[4] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[5] R. Chatterjee, B.G. Ryder, and W. Landi, "Relevant Context Inference," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 133-146, 1999.

[6] B. Cheng and W. Hwu, "Modular Interprocedural Pointer Analysis Using Access Paths," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 57-69, 2000.

[7] M. Das, "Unification-Based Pointer Analysis with Directional Assignments," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 35-46, 2000.

[8] M. Emami, R. Ghiya, and L. Hendren, "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 242-257, 1994.

[9] M. Fähndrich, J. Foster, Z. Su, and A. Aiken, "Partial Online Cycle Elimination in Inclusion Constraint Graphs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 85-96, 1998.

[10] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo, "Points-To Analysis for Program Understanding," *J. Systems and Software,* vol. 44, no. 3, pp. 213-227, Jan. 1999.

[11] S. Ghosh, N. Bawa, S. Goel, and Y.R. Reddy, "Validating Run-Time Interactions in Distributed Java Applications," *Proc. IEEE Int'l Conf. Eng. Complex Computer Systems,* pp. 7-16, 2002.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification,* third ed. Addison-Wesley, 2005.

[13] W. Grosso, *Java RMI.* O'Reilly, 2002.

[14] D. Grove and C. Chambers, "A Framework for Call Graph Construction Algorithms," *ACM Trans. Programming Languages and Systems,* vol. 23, no. 6, pp. 685-746, Nov. 2001.

[15] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call Graph Construction in Object-Oriented Languages," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 108-124, 1997.
[16] B. Haumacher and M. Philippsen, "Exploiting Object Locality in JavaParty, a Distributed Computing Environment for Workstation Clusters," *Proc. Workshop Compilers for Parallel Computers,* pp. 83-94, June 2001.
[17] N. Heintze and O. Tardieu, "Ultra-Fast Aliasing Analysis Using CLA: A Million Lines of C Code in a Second," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 254-263, 2001.
[18] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?" *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.,* pp. 54-61, 2001.
[19] W. Landi and B.G. Ryder, "A Safe Approximation Algorithm for Interprocedural Pointer Aliasing," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 235-248, 1992.
[20] O. Lhoták, "Spark: A Scalable Points-to Analysis Framework for Java," master's thesis, McGill Univ., Dec. 2002.
[21] O. Lhoták, L. Hendren, "Scaling Java Points-To Analysis Using Spark," *Proc. Int'l Conf. Compiler Construction,* pp. 153-169, 2003.
[22] O. Lhoták and L. Hendren, "Context-Sensitive Points-to Analysis: Is It Worth It?" *Proc. Int'l Conf. Compiler Construction,* 2006.
[23] D. Liang, M. Pennings, and M.J. Harrold, "Extending and Evaluating Flow-Insensitive and Context-Insensitive Points-to Analyses for Java," *Proc. Workshop Program Analysis for Software Tools and Eng.,* pp. 73-79, June 2001.
[24] D. Liang, M. Pennings, and M.J. Harrold, "Evaluating the Impact of Context-Sensitivity on Andersen's Algorithm for Java Programs," *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.,* 2005.
[25] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient Java RMI for Parallel Programming," *ACM Trans. Programming Languages and Systems,* vol. 23, no. 6, pp. 747-775, Nov. 2001.
[26] *RMI Specification,* Sun Microsystems, 2002.
[27] *Serialization Specification,* Sun Microsystems, 2003.
[28] A. Milanova, A. Rountev, and B.G. Ryder, "Parameterized Object Sensitivity for Points-To and Side-Effect Analyses for Java," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 1-11, 2002.
[29] A. Milanova, A. Rountev, and B.G. Ryder, "Parameterized Object Sensitivity for Points-To Analysis for Java," *ACM Trans. Software Eng. and Methodology,* vol. 14, no. 1, pp. 1-41, Jan. 2005.
[30] M. Philippsen and B. Haumacher, "Locality Optimization in JavaParty by Means of Static Type Analysis," *Concurrency: Practice and Experience,* vol. 12, no. 8, pp. 613-628, July 2000.
[31] M. Philippsen, B. Haumacher, and C. Nester, "More Efficient Serialization and RMI for Java," *Concurrency: Practice and Experience,* vol. 12, no. 7, pp. 495-518, May 2000.
[32] B. Quig, J. Rosenberg, and M. Kölling, "Supporting Interactive Invocation of Remote Services," *Proc. Int'l Conf. Principles and Practice of Programming in Java,* pp. 195-200, 2003.
[33] C. Razafimahefa, "A Study of Side-Effect Analyses for Java," master's thesis, McGill Univ., Dec. 1999.
[34] A. Rountev, "Component-Level Dataflow Analysis," *Proc. Int'l SIGSOFT Symp. Component-Based Software Eng.,* pp. 82-89, 2005.
[35] A. Rountev and S. Chandra, "Off-Line Variable Substitution for Scaling Points-To Analysis," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 47-56, 2000.
[36] A. Rountev, S. Kagan, and T. Marlowe, "Interprocedural Dataflow Analysis in the Presence of Large Libraries," *Proc. Int'l Conf. Compiler Construction,* pp. 2-16, 2006.
[37] A. Rountev, A. Milanova, and B.G. Ryder, "Points-To Analysis for Java Based on Annotated Constraints," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 43-55, Oct. 2001.
[38] A. Rountev and B.G. Ryder, "Points-To and Side-Effect Analyses for Programs Built with Precompiled Libraries," *Proc. Int'l Conf. Compiler Construction,* pp. 20-36, 2001.
[39] E. Ruf, "Effective Synchronization Removal for Java," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 208-218, 2000.
[40] B.G. Ryder, "Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages," *Proc. Int'l Conf. Compiler Construction,* pp. 126-137, 2003.
[41] B.G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher, "A Schema for Interprocedural Modification Side-Effect Analysis with Pointer Aliasing," *ACM Trans. Programming Languages and Systems,* vol. 23, no. 2, pp. 105-186, Mar. 2001.
[42] M. Shapiro and S. Horwitz, "Fast and Accurate Flow-Insensitive Points-to Analysis," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 1-14, 1997.
[43] M. Sharp and A. Rountev, "Static Analysis of Object References in RMI-Based Java Software," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 101-110, 2005.
[44] B. Steensgaard, "Points-to Analysis in Almost Linear Time," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 32-41, 1996.
[45] M. Streckenbach and G. Snelting, "Points-To for Java: A General Framework and an Empirical Comparison," technical report, Univ. of Passau, Sept. 2000.
[46] Z. Su, M. Fähndrich, and A. Aiken, "Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 81-95, 2000.
[47] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo, "Flow Insensitive C++ Pointers and Polymorphism Analysis and Its Application to Slicing," *Proc. Int'l Conf. Software Eng.,* pp. 433-443, 1997.
[48] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo, "Variable-Precision Reaching Definitions Analysis," *J. Software Maintenance: Research and Practice,* vol. 11, no. 2, pp. 117-142, Mar. 1999.
[49] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?," *Proc. Int'l Conf. Compiler Construction,* pp. 18-34, 2000.
[50] R. Veldema and M. Philippsen, "Compiler Optimized Remote Method Invocation," *Proc. IEEE Int'l Conf. Cluster Computing,* pp. 127-137, 2003.
[51] J. Whaley and M. Lam, "Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams," *ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2004.
[52] S. Yong, S. Horwitz, and T. Reps, "Pointer Analysis for Programs with Structures and Casting," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 91-103, 1999.
[53] S. Zhang, B.G. Ryder, and W. Landi, "Program Decomposition for Pointer Aliasing: A Step towards Practical Analyses," *Proc. ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 81-92, 1996.

Mariana Sharp is a PhD candidate in the Department of Computer Science and Engineering at the Ohio State University. Her research interests include program analysis of object-oriented and distributed software. She is a student member of the IEEE Computer Society and the ACM.

Atanas Rountev received the PhD degree in computer science from Rutgers University. He is an assistant professor in the Department of Computer Science and Engineering at the Ohio State University. His research interests include static and dynamic program analysis, software understanding and evolution, software testing, component-based software, distributed software, and high-performance computing. His recent research focuses on scalable modular program analyses, on algorithms and tools for reverse engineering, and on compile-time and runtime support for scientific computing. Dr. Rountev has served on the program committees of several conferences and workshops, including the International Conference on Software Engineering (ICSE), the International Symposium on Software Testing and Analysis (ISSTA), and the International Conference on Software Maintenance (ICSM). He is a member of the IEEE Computer Society, the ACM, SIGSOFT, and SIGPLAN.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.