Object Graph Analysis

André Spiegel

spiegel@inf.fu-berlin.de

Technical Report B–99–11
July 1999

Abstract

The run-time structure of an object-oriented program can be represented by an object graph. Approximating this graph statically is a prerequisite for higher level analyses such as distribution analysis and concurrency analysis; it is also helpful in contexts of software maintenance and re-engineering. However, most existing techniques for static analysis of object-oriented programs are not adequate for deriving general object graphs from the source code. We have therefore developed a new algorithm that is capable of doing so. The algorithm is defined for the Java language, of which it covers all language features except class loader interactions and run-time reflection. It is flow-insensitive but context-sensitive, and therefore has a low computational complexity. This paper describes the algorithm and presents results that our implementation obtained for several non-trivial example programs of considerable size.

Keywords: Static analysis, run-time structure, object graph, Java.

Freie Universität Berlin
Institut für Informatik
Takustraße 9
D–14195 Berlin, Germany

1 Introduction

For object-oriented programs, static analysis typically answers questions such as: what is the run-time type of an expression that appears in the code [PS91, PC94]; where does a certain pointer variable point to at run-time [Ste96, SH97, CRL99]; what method will actually be called in a dynamically dispatched call [GDDC97]. This information is useful, either directly or indirectly, for compiler optimizations such as static binding of method calls.

However, this is a fairly traditional view of an object-oriented program. It considers the program to be a static sequence of statements, grouped in procedures (methods), manipulating a passive data structure on the heap (the objects). This is not the view that programmers are trained to have. For them, a program at run-time consists of a *set of objects* that interact with each other by invoking methods or accessing fields; following either a single or multiple flows of control. Some of the program's objects might happen to share the same code; nonetheless the programmer views them as separate entities.

This is often more than a philosophical issue. For example, when programs are distributed across multiple machines, the unit of distribution is generally the object, not the class. Program understanding, as required for software maintenance and re-engineering, is another area where an object-oriented view of a program is needed.

In this paper, we present an algorithm that analyzes the source code of a Java program to derive an object graph from it, representing the program's run-time structure. We have developed this algorithm as part of a system that can distribute Java programs automatically [Spi00a, Spi00b]. However, it has turned out as a very useful tool for program understanding as well, allowing the programmer to quickly grasp the essential structure of a program before even looking into the source code. In addition, we believe our algorithm is a vital contribution to all sorts of analyses where the storage structure of an object-oriented program is sought, e.g. concurrency analysis (model checking) [Cor98].

The result of our algorithm is a graph, the nodes of which represent the objects that will exist at run-time, with three kinds of edges between them: creation edges, reference edges, and usage edges. The algorithm *approximates* the actual run-time structure in that (a) some nodes in the object graph may be summary nodes that represent zero to many actual run-time objects, (b) reference edges and usage edges are conservative, and (c) at least in the final object graph, we consider objects as unstructured containers of references, abstracting from their internal structure.

According to the way in which analysis algorithms are usually classified (as in e.g. [SH97]), our algorithm is largely *flow-insensitive* (it does not consider the actual flow of control within methods), it is however *context-sensitive* in the sense that method invocations and field accesses are distinguished at the level of objects, not types. Due to its flow-insensitivity, the algorithm is of low computational complexity (essentially polynomial). Initial experience with our implementation shows that non-trivial real-world programs can thus be analyzed in acceptable time; detailed results are given below.

This paper is organized as follows. In section 2, we review existing techniques for static analysis of object-oriented programs, showing that they are not sufficient for constructing general object graphs. Section 3 describes our algorithm, and section 4 discusses its complexity. In section 5, we show the object graphs of three example programs, and present quantitative results for several others. Section 6 discusses possible improvements of the algorithm, and section 7 concludes the paper.

```
     public class Main {
       ...
       public static void main (...) {
1:       Worker w1 = new Worker();
2:       Worker w2 = new Worker();
3:       w1.doWork(); ... w2.doWork();
            // maybe in parallel
         ...
      }
    }

    public class Worker {
4:    Algorithm a;
5:    public Worker() {
6:      a = new Algorithm();
      }
7:    public void doWork() {
8:      while (...)
9:        a.calculate (...);
      }
    }

    public class Algorithm {
10:   public void calculate (...) { ... }
    }
```

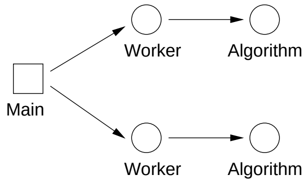


Figure 1: Example Program and Corresponding Object Graph

2 Related Work

Our work is related to, but distinct from, existing techniques for *call graph construction*, *concrete type inference*, and *points-to analysis* of object-oriented programs. There is considerable overlap between these areas, e.g. points-to analysis often involves some kind of call graph construction and vice versa, yet the headings under which we discuss them do indicate the primary focus of the corresponding research.

Call graph construction ([GDDC97] and references therein) is concerned with finding, for each call site in a program, the corresponding set of methods that may be invoked at run-time, the goal being compiler optimizations such as static binding of method invocations, or to enable method inlining and further interprocedural analyses. Call graphs, as introduced in the literature, are distinct from object graphs in that they refer to the static types of a program, not the dynamic instances. As an example for this difference, consider the program in Figure 1, where two `Worker` objects each use an `Algorithm` object to perform a calculation. A call graph for this program is easy to construct, as all calls are monomorphic: the call sites in line 3 refer to `Worker.doWork()` in line 7, while in line 9, `Algorithm.calculate()` in line 10 is called. This call graph, while it is sufficient for the kinds of compiler optimization that we mentioned above, does not, however, imply the run-time structure of the program in the sense that is shown in figure 1, which represents the relations between *instances*. For this information to be captured in a call graph, a notion of calling context that includes the identity (not the type) of the implicit `this` parameter would be required. We are not aware of any such approach.

Concrete type inference [PS91, PC94, GI98] subsumes techniques to derive, for expressions appearing in the program code, precise type information, thus potentially reducing or eliminating polymorphism, and thereby enabling compiler optimizations similar to those mentioned above. It is clear, however, that this also does not capture relations between *instances*: in the example program, all types are easily resolved monomorphically, but this only represents the fact that, e.g. `Worker` objects access `Algorithm` objects, but not their numbers and one-to-one correspondence.

Based on this observation, an extension of type inference has been described by Philippsen and Haumacher [PH98]. In their algorithm, *helper polymorphism* is introduced into programs in order to make the types of separate instances distinct, so that traditional type inference can then be used to yield instance-level structure (in this case, for the purpose of locality optimization in concurrent Java programs). The technique is, however, only applied to `Threads` and `Runnable` objects. If it were extended to the general case, it could yield similar results to the algorithm we present here. A difference is however that the algorithm of Philippsen and Haumacher is flow-sensitive, and thus problematic for larger programs (very long running times are reported in [Hau98]). It is also questionable whether the attempt to create, ideally, a separate type for each instance, is conceptually sound, as it blurs the otherwise useful distinction between types and instances.

A third large body of research is subsumed by the term *points-to analysis.* Citing [CRL99], the goal of points-to analysis is *to determine, at each program point, the objects to which a pointer may point during execution.* Some of this research is stack-oriented (e.g. [And94, Ste96, SH97]), i.e. it only considers pointer variables on the stack (whether they point to the stack or to heap-allocated storage), but not pointers between objects on the heap, and thus it is not immediately relevant for our purposes.

In [CRL99], an approach called *Relevant Context Inference (RCI)* is described, which extends traditional stack-oriented techniques towards object-oriented programming, i.e. to general pointer structures between heap-allocated objects. A closer look, however, reveals that RCI does not accurately provide the information we are interested in. The main reason for this is that RCI identifies object allocations and pointer expressions *by their textual location* in the program code, not parameterized by the instance they appear in at run-time. In our example program in listing 1, RCI would summarize the two `Algorithm` objects into a single node "objects created at line 6", and thus lose their identity. While it is true that our own algorithm uses summary nodes frequently as well, it folds the object graph to a much lesser degree than RCI does.

Also subsumed under the term *points-to analysis*, other research has focused entirely on heap-allocated data structures [CWZ90, VHU92, SRW96, Cor98]. The common methodology of these analyses is to perform an abstract interpretation of the program code and to construct, for each statement, a *storage structure graph* that represents the possible heap structures at that statement. Often, this approach is used to allow some kind of *shape analysis* of the heap structure, e.g. to prove that, if a procedure receives a list-like data structure, it preserves the property of list-ness during execution [CWZ90, SRW96].

More closely related to our work, Vitek *et al.* [VHU92] and Corbett [Cor98] have applied the above approach to object graphs of complete programs. Both algorithms are flow-sensitive; they may thus provide higher accuracy than our algorithm at the cost of prohibitive performance for larger programs. The algorithm of Vitek *et al.* is defined for a Smalltalk-like toy language, while Corbett's algorithm is part of a model checker for concurrent Java programs. No implementation or performance figures are reported for either algorithm. It must also be noted that both approaches suffer from their heritage of traditional, non-object-oriented program analysis: they maintain the notion of a *static program code* that manipulates a *passive data structure* on the heap. One of the results of this is that the analysis of polymorphic method calls becomes more complicated than it could be if the objects were considered "active" first-class entities, as in our approach. To reduce polymorphism, Vitek *et al.* employ the common, type-based technique of including k levels of the dynamic call chain as context information. In Corbett's algorithm, on the other hand, all method calls must be *inlined* prior to the actual analysis (implying that recursion cannot be analyzed), and polymorphism is accounted for by simply inlining the code of *all* corresponding method implementations in a `switch` statement – which, as the author acknowledges, results in an exponential complexity and also forfeits much of the precision that a flow-sensitive algorithm could otherwise have.

By contrast, our own algorithm is flow-insensitive (and thus applicable to large programs), and based entirely on the notion of *objects* which, at run-time, organize themselves into an object graph. We will now describe this algorithm in detail.

3 The Algorithm

The entities that our algorithm deals with are the *types* of a program, and the *objects* that these types are instantiated to at run-time. Our model of these entities – the "ontology" of our algorithm – is shown in Figure 2.

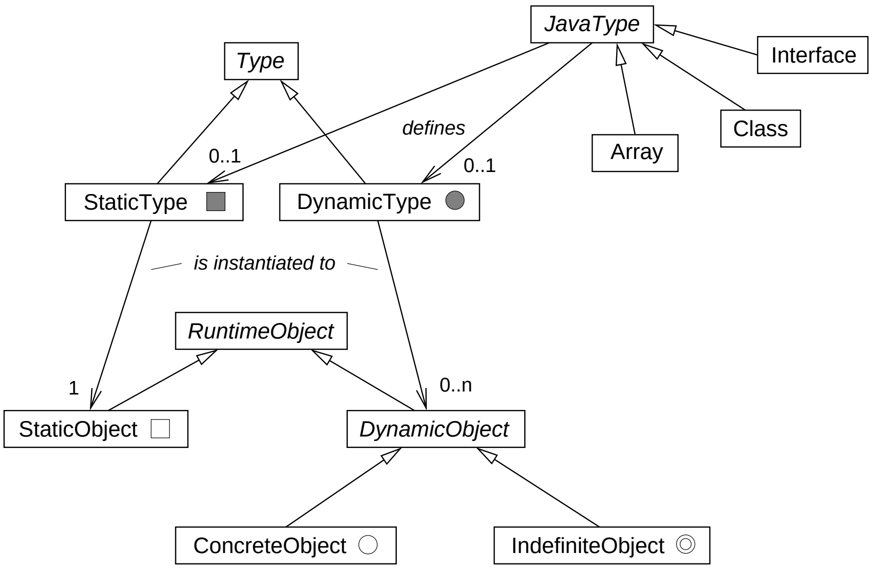


Figure 2: Ontology

In Java, objects are defined by *reference types*, which can be classes, interfaces, or arrays. As we are not concerned with primitive types, we will also use the word *Java type* as a shorthand for classes, interfaces, and arrays.

Java types may have "static" and "non-static" members. To deal with this distinction in a natural way, it is helpful to introduce a slightly different type model for analysis purposes: we say that a Java type defines, optionally, both a *static type* (comprising the static members) and a *dynamic type* (the non-static members). We consider these types, and their instances, entirely separate entities[1]. Of a static type, precisely one instance

[1]There is, in fact, no special connection between the instances of a Java class and the static members of that class, as compared to static members of other classes. The instances do have privileged access to any *private* members of their class, but this is only a question of accessibility and not important for our analyses.

exists at run-time (a static type is pretty much the same as a module), while a dynamic type may have an arbitrary number of instances. We use the common term *analysis type*, or simply *type*, to refer to both static types and dynamic types in the following.

At run-time, types are instantiated to objects. We call the single instance of a static type a *static object*, and the instances of a dynamic type *dynamic objects*. Of the latter, there are two further subcategories: a *concrete object* represents a single instance of a dynamic type, the existence of which at run-time is certain. An *indefinite object*, on the other hand, summarizes 0 to n objects of a dynamic type; the algorithm cannot determine their precise number. (Note, though, that the use of indefinite objects does not mean that the algorithm degenerates into a mere type-based analysis: for a given dynamic type, several indefinite objects may exist in an object graph; each represents those instances of the type that occur in a certain context.)

The *relations* between objects that we are interested in are *creation*, *reference*, and *usage*. We say that

- an object a *creates* an object b if the statement by which object b is allocated is executed in the context of object a;
- an object a *references* another object b if, at any time during execution, a reference to b appears in the context of a (either in a field, variable, or parameter, or as the actual value of an expression; we also say that *a owns a reference to b* or simply that *a knows b*);
- an object a *uses* an object b if a invokes any methods or accesses any fields of b.

It is clear that *usage* implies *reference*, because an object can only *use* another object if it owns a reference to it, but not vice versa (e.g. a collection object owns references to the objects contained in it, but does not usually invoke any methods of these objects). Similarly, *creation* usually implies *reference*, because an object that creates another object immediately receives a reference to it. (An exception are object allocations that occur as actual method parameters; some of these cases are recognized by the algorithm, see section 3.3 for details.)

The object graph is constructed in the following steps:

Step 1. Find the *set of types* that the program consists of.

Step 2. Build a *type graph* from these types, which captures usage relations and data flow relations at the type level.

Step 3. Approximate the *object population* of the program, which yields the nodes of the object graph, plus creation edges and initial reference edges.

Step 4. *Propagate references* in the object graph, based on the data flow information from the type graph.

Step 5. Create *usage edges* in the object graph.

We will now describe each of these steps in detail.

3.1 Finding the types of the program

The Java types that a program consists of are those contained in the *dependency closure* of the program's main class. We say that a Java type *depends* on another type if it makes any kind of syntactic reference to it (an obvious exception being class `java.lang.Object`, which is part of every program although it needn't be referred to explicitly). The set of Java types naturally implies the set of *analysis types* of the program, according to the ontology described above.

This definition ensures what may be called the *closed-world assumption* of our algorithm: at run-time, control cannot reach any statement that is not covered by the static dependency closure.

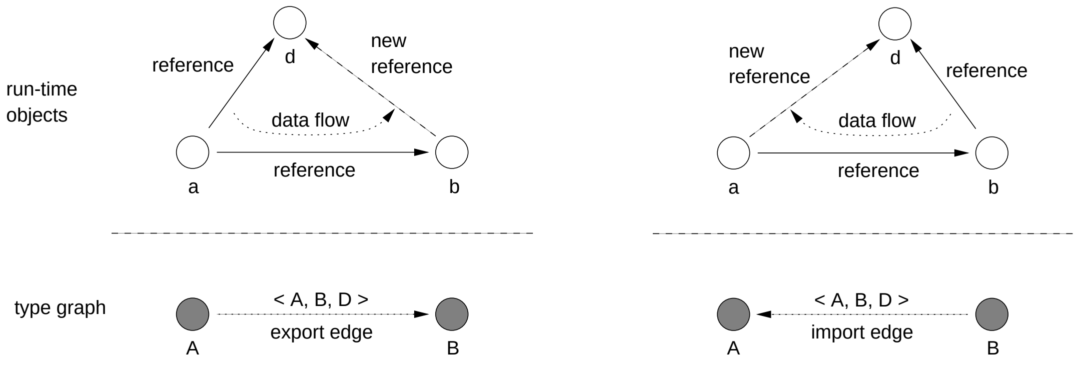
It must be noted, though, that Java programs can dynamically modify and extend themselves through explicit class loader interaction and run-time reflection. Naturally, the use of these features poses a whole set of new problems for any static analysis algorithm. We are not adressing these in our work, and our algorithm cannot handle programs that make use of these features. At present, this does not seem like a serious limitation, as few programs actually fall into this category. Future research in this area is however desirable.

Our algorithm is also restricted to analysis of complete programs; we have not investigated techniques to analyze libraries, and to combine such analyses incrementally when analyzing programs that use these libraries.

3.2 Constructing the type graph

Ultimately, we are interested in the run-time objects of the program and their relations. However, what we have so far is only the set of types from which the objects will be instantiated. Our next step is therefore to analyze some relations *at the type level*, capturing them in a *type graph*, which will later be used when we construct the actual object graph.

A relation between two types is a *folding* of the relations between any objects that are instantiated from these types. To deal with this folding, a natural shorthand terminology will be used in the following: we say that "a type A calls a method of another type B" if the code of A contains a method call statement, the syntactic target of which is a method declared in type B. As our algorithm is flow-insensitive, the existence of such a statement is enough for us to conclude that at run-time, any object that is instantiated from A might call any object instantiated from B (subtyping will be dealt with at a later stage). An analogous definition holds for expressions such as "type A accesses a field of type B", etc.



(a) Export (b) Import

Figure 3: Data Flow between Objects

Given this, we can define the type graph as a directed graph, $G_t =< T, E_u, E_e, E_i >$, where T is the set of types that we computed in the first step, and E_u, E_e, and E_i are *usage edges*, *export edges*, and *import edges*, respectively.

A *usage edge* $< A, B >_u$ simply means that type A *uses* type B, i.e. it calls methods or accesses fields of type B.

Export edges and *import edges* are data flow edges which indicate that references of a certain type may propagate from objects of one type to objects of another type, e.g. as parameters of method calls or by direct field accesses. We distinguish two fundamental kinds of such reference propagation (see fig. 3): to *export* a reference means that an object a owns a reference to an object d, and passes it to an object b. To *import* a reference means that an object a owns a reference to an object b, from which it receives a reference to an object d which only b knew before.

In the type graph, we represent this by export edges and import edges from a type A to a type B, annotated by a third type D, which is the type of the data. The type graph contains an *export edge* $< A, B, D >_e$ if

- A calls a method of B, and at least one of the actual parameters of this method call is of type D, or
- A assigns (writes) to a field of B, and the actual r-value of the assignment is of type D, or
- B is an array and A assigns (writes) references to objects of type D into B,

and there is an *import edge* $< A, B, D >_i$ if

- A calls a method of B that has D as its declared return type, or
- A reads a field of B, the declared type of which is D, or
- B is an array with element type D and A reads elements of B.

Two remarks about these definitions. First, they imply that we do not give full object status to *exceptions*, although they might, technically, be used to carry (and hence, pass) objects of arbitrary types as a payload. However, this is not a principal restriction; see section 6 for a discussion of alternatives.

Second, a note about subtyping and polymorphism. In principle, we do not need to consider these at this stage of the algorithm, because the information can easily be inferred conservatively from the type graph constructed so far. For example, an export edge $< A, B, D >_e$ that we found syntactically implies analogous export edges for all subtypes of A, B, and D within the program. In other words, if an object of type A may export references of type D to objects of type B, then any object of any subtype of A might also export references of type D or any subtype of D to any object of type B or any subtype of B. It is however a pure implementation issue whether we actually insert additional data flow edges to cover these cases, as they are completely redundant. In our implementation, we chose to propagate edges to subtypes on both their source and destination side (A and B), in order to speed up subsequent interpretation of the graph (in step 4 of the algorithm), but we do not create additional edges for subtypes of the data types (D) in order not to use too much memory.

We may now proceed to construct the *object graph*, which is a directed graph $G_o =< O, E_c, E_r, E_u >$, where O is the set of run-time objects of the program (we also call it the *object population*), and E_c, E_r, and E_u are sets of creation edges, reference edges, and usage edges, respectively.

In step 3 of the algorithm (section 3.3), the object population is constructed, using indefinite objects (summary nodes) where necessary and concrete objects where possible. In step 4 (section 3.4) the reference structure within the object population is computed, and in step 5 (section 3.5), usage relations are inferred.

3.3 Generating the object population

The object population of a program is a complete, but finite representation of the potentially infinite set of objects that the program will create at run-time. In the terms of our algorithm, the object population is a set of static objects, concrete objects and indefinite objects, which form the nodes of the object graph. The algorithm constructs this set by examining the object allocation statements[2] in the program, determining which objects may (or definitely will) create which other objects.

We distinguish two kinds of allocation statements: an *initial allocation* is an object allocation that is executed exactly once whenever the enclosing type is instantiated, and never thereafter (in the context of this particular object). A *non-initial allocation*, on the other hand, is an object allocation of which the algorithm cannot determine how often, if ever, it will be executed at run-time.

Our algorithm considers an allocation as *initial* for its enclosing type A if

- it is the r-value of a *field initializer* of A, or
- it occurs plainly in an initialization method of A, where *plainly* means that it is not nested in any kind of control structure, and an *initialization method* is defined as either

[2] An object allocation is a `new` expression in Java. Another way to create an object is to `clone()` another object; in this case, the precise type of the new object may not be known statically. Conservatively, a call to `a.clone()` can be considered to allocate an indefinite object of type A and every subtype of A in the program.

- the constructor[3] of A, or
- a static initializer of A, or
- the `main()` method, if A is the program's static main type (and there is no explicit call to `main()` within the program), or
- the `run()` method, if A is a `Runnable` object (and there is no explicit call to `run()` in the program), or
- a private method of A that is called exactly once and plainly from another initialization method of A.

Based on this definition, the algorithm can compute, recursively, which objects create which other objects. It begins by adding the static objects of the program (one static object for each static type is trivially part of the object population), and then proceeds as follows:

- For each *static object* of a type A that is added to the object population, the algorithm adds one concrete object for each initial allocation of A, and one indefinite object for each type that is non-initially allocated in A.
- For each *concrete object* of a type A that is added to the object population, concrete objects and indefinite objects are added in the same manner as for a static object.
- For each *indefinite object* of a type A that is added to the object population, all allocation statements in A are treated as non-initial allocations, i.e. for each type that is allocated in A, an indefinite object is added to the object population.

Intuitively, the above means that static objects and concrete objects may recursively create further concrete objects – those that they allocate initially. Indefinite objects, however, may only create further *indefinite* objects (because it is not known how many objects an indefinite parent object actually represents). Also, note that the non-initial allocations of a type are summarized by the types being allocated. An indefinite object therefore represents all instances of a certain type that may be created by a given parent object, excluding any concrete objects of the same type that were created by that parent. (We chose not to distinguish individual allocation statements within the parent type for indefinite objects, because that keeps the size of the object graph somewhat smaller. We have found this to produce adequate results for our purposes.)

Whenever a concrete object or indefinite object is added to the graph, we also add both a creation edge and a reference edge from the parent object to the new object. In the next step, these initial references will be propagated within the graph to determine which objects may know and use which other objects at run-time. However, there are three special cases that need to be considered.

First, static objects are referred to by *name* in Java, not by object references. To deal with this in a uniform way, we therefore add "pseudo" reference edges from each dynamic object to any static objects it *uses*, according to the type graph.

[3] Dynamic types with multiple constructors may have different sets of initial allocations for each constructor, and the chaining of constructors along the inheritance hierarchy also needs to be considered. The details are straightforward; we are omitting them here for brevity.

Second, if an object allocation appears directly as an actual method parameter, then the creator does not actually receive a reference to the created object. However, at this stage of the algorithm it is not usually known which object is actually called and receives the reference. The only exception are allocations that are used directly as *constructor parameters*, because here the receiver is immediately known. The algorithm adequately handles this case, in all other cases, we conservatively consider the creator to own the reference, which is later propagated to the possible receivers in step 4.

A third special case that needs to be considered are *cycles* in the creation structure: if an indefinite object allocates an indefinite object that has the same type as one of its (transitive) parent objects, the algorithm would not terminate, and create an infinite amount of objects. (For concrete objects, this cannot happen because these are allocated *initially* – a cycle here would mean that the program itself falls into endless recursion immediately after startup.) For indefinite objects, the algorithm recognizes cycles by keeping track of all parents for each indefinite object. If a cycle is detected for a type A, the algorithm does not add a further indefinite object of type A, but rather adds creation and reference edges *back* to the existing parent of type A, and terminates the recursion.

3.4 Propagating References

After the object population has been computed, the object graph contains the representation of all objects that could possibly exist at run-time, connected by creation edges and reference edges. We now use the data flow information from the type graph to propagate the references edges within the object graph until a fix point is reached.

The actual algorithm corresponds exactly to the scenarios shown in Fig. 3: it iterates over all triples of objects $< a, b, d >$ for which reference edges $< a, b >_r$ and $< a, d >_r$ (or $< b, d >_r$) exist, and matches the types of the objects against the data flow edges of the type graph. If a corresponding edge exists, a new reference $< b, d >_r$ (or $< a, d >_r$) is added to the graph.

It is here that subtyping must be accounted for. If the type graph did not contain redundant data flow edges for subtypes (see section 3.2), we'd have to search it for data flow edges of the form $< A, B, D >$ where A, B, and D are the types *or any supertypes* of a, b, and d. But as our type graph has been constructed to contain redundant edges for all subtypes of sources and destinations already, we only need to search for supertypes of D, which is further simplified because in the implementation, we combine all edges between two types into a single edge annotated with a *set* of data types.

3.5 Adding usage edges

After the object references have been propagated, it is known which object could possibly interact with which other objects. We may now add usage edges to the graph: there is a usage edge between two objects a and b if there is a usage edge $< A, B >_u$ between their types A and B in the type graph, *and* there is a reference edge $< a, b >_r$ in the object graph.

As in the previous step, subtyping is adequately considered here. When the type graph was constructed (see section 3.2), redundant usage edges were added for all known subtypes on both the source and destination side. In other words, this means that if an object a of type A knows an object b of type B, and A or any supertype of A uses B or any supertype of B, then the object a is considered to use object b.

4 Complexity

Let s be the number of statements in the program, t the number of types in the program, and n the number of run-time objects (the size of the object population computed in step 3). The first and the second step of the algorithm are uncritical: the first step – finding the types of the program – only involves standard syntactic type inference (better than $O(s^2)$), and the second step (construction of the type graph) is linear in s.

Constructing the object population (in step 3) is linear in n, the number of run-time objects needed. It is not obvious how this number relates to the static size of the program. The worst case occurs when only concrete objects are used, as each concrete object could allocate an arbitrary number of further concrete objects, provided that types of parent objects are not used again and that each such allocation occurs due to an individual statement in the code. In a program with t types, each of which contains s/t initial allocation statements, the size of the object graph is thus $(s/t)^t$.

This exponential complexity is however unlikely to occur in practice. In real programs, concrete objects represent the static part, or "skeleton" of the run-time structure, which is usually small, while everything that depends on input data or user interaction is modelled by indefinite objects. In the eleven programs discussed in section 5.4, there is in fact a roughly *linear* correspondence between s and n: the final object graphs contain about one object (static, concrete, or indefinite) for every 10–50 lines of source code.

In step 4, references are propagated among triples of objects using fix-point iteration (similar to computing the transitive closure). Our algorithm is optimized in that it only considers those references that were created in the previous step for further propagation. For each reference, this requires work that is linear in n, and since at most n^2 reference edges may exist in the graph, the entire step has complexity $O(n^3)$.

Step 5, the creation of usage edges, is again uncritical: checking whether a usage edge is needed between two objects, and possibly creating the edge, requires essentially constant time, and it must be done for each pair of objects connected by a reference edge, which is at most n^2 times.

The overall complexity of the algorithm is thus $O(n^3)$, where n appears to be linear in the size of the program s for real-world programs.

5 Case Studies

We will illustrate the kinds of results that our algorithm delivers in a number of case studies now. We will look in detail at the object graphs of three small to moderately sized programs (sections 5.1, 5.2, and 5.3), and discuss the performance of our implementation in section 5.4.

5.1 Case Study 1: Producer/Consumer

Figure 4 shows the object graph of a simple producer/consumer program. This program is the same that Corbett analyzed in his paper [Cor98], with one added complexity: rather than passing primitive integers from the producer to the consumer, we modified the program to use `Integer` objects, so that they would be visible in the object graph. The complete code of the program is listed in appendix A.

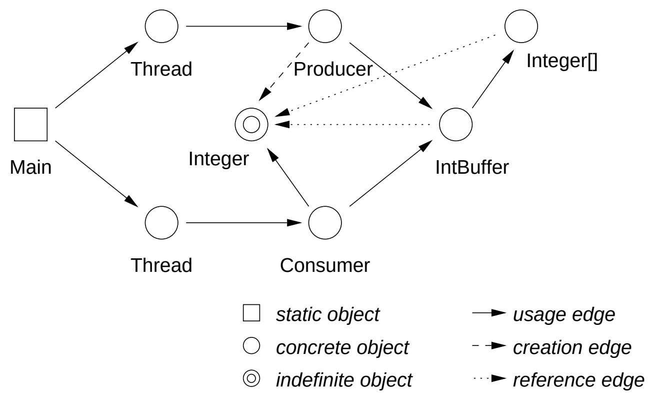


Figure 4: Object Graph of the Producer/Consumer Program

The structure of the program is immediately clear from the graph: there is a `Producer` and a `Consumer` object, which both implement the `Runnable` interface; they are executed in parallel by corresponding `Thread` objects. `Producer` and `Consumer` share the `IntBuffer` object, through which the `Integer` objects are passed. Internally, the `IntBuffer` stores the `Integer` objects in an array. For clarity, we have omitted the creation edges for all objects except for the `Integer` objects, and one reference edge from `Main` to the `IntBuffer` object. Each usage edge shown naturally implies a reference edge.

All objects in the graph are concrete, i.e. it is certain that only one instance of them will exist at run-time, except for the `Integer` objects, which are created in arbitrary numbers by the `Producer`. They are referenced (but not used) by the `IntBuffer` and its internal array; the `Consumer` does use them (extracting the integer value and printing it).

5.2 Case Study 2: Hamming's Problem

Hamming's problem is one of the four Salishan problems, a suite of typical parallel programming problems often used to compare the expressiveness of parallel programming languages. For Hamming's problem, the task is to output a sorted sequence of integers of the form p^i, where $i = 0, 1, 2, ...$ and p is any of a given set of prime numbers $\{a, b, c, ...\}$. The parallel implementation is to have one thread for each of the primes, which computes the p^i values for that prime. All threads deliver their results to a centralized manager which selects the next number for the sequence among them.

Figure 5 shows the object graph of a Java implementation of this algorithm. The size of the program is about 170 LOC; the object graph is computed in 6 seconds (see section

5.4 for detailed results). All creation edges have been omitted from the graph; all usage edges and reference edges are shown (each usage edge implies a reference edge).

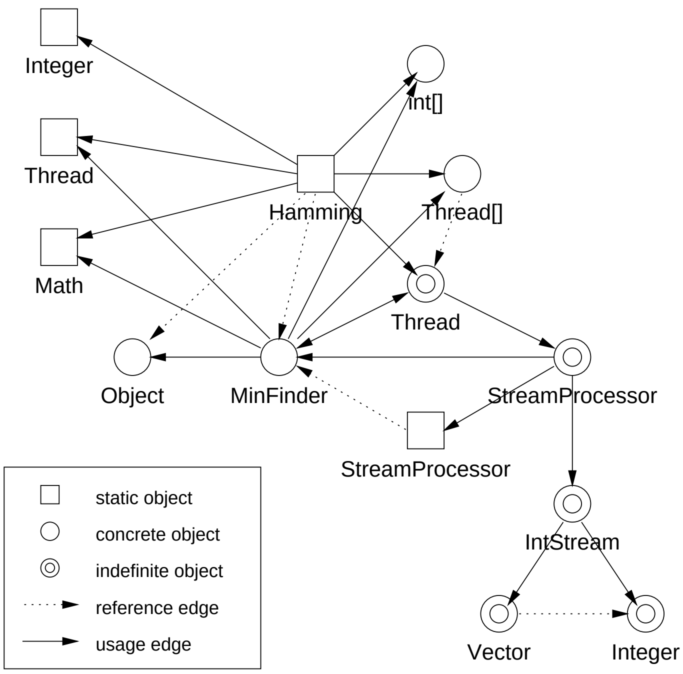


Figure 5: Object Graph for Hamming's Problem

The graph shows an indefinite number of `StreamProcessor` objects (one for each prime). The sequence of numbers produced by each is internally stored in an `IntStream` queue, which in turn is realized as a `Vector` of `Integer` objects. All `StreamProcessors` use a common `MinFinder` object to which they deliver their results (the reference to the `MinFinder` is obtained from a static variable in the `StreamProcessor` class). Internally `MinFinder` uses an instance of `java.lang.Object` for synchronization purposes.

A non-trivial property of this object graph is that polymorphism in the `Vector` class is adequately analyzed here: as with all Java collection classes, the element type of `Vector` is `Object`, i.e. anything could be stored in a `Vector`. However, due to the way we compute the reference structure within the graph, we can infer correctly that in these `Vector` objects, only the `Integer` objects created by the corresponding `IntStream` are stored. We have found that even in large programs where many different collections for different actual element types are used, the algorithm usually determines correctly which objects are stored where.

However, there is also a counter-example in this graph. The object used for synchronization by the `MinFinder` is *actually* of type `java.lang.Object`. This object is created by the main `Hamming` object, and passed to the `MinFinder`. Since the algorithm is flow-insensitive, it must be assumed that *any* object could be passed along this edge, and therefore the `MinFinder` receives references to all objects that `Hamming` knows, i.e. also the indefinite `Thread` object and the two arrays (static objects are not passed). Furthermore, since `MinFinder` invokes methods of the `Object` instance, it must also be assumed that it uses all other objects that it knows, including the ones that were mistakenly passed to it in the previous step. The graph thus contains several spurious reference and usage edges.

Despite this imprecision, the object graph is not only useful for understanding the program, but also for deciding on a distribution policy. The `StreamProcessor` objects, although their actual number is not known statically, can be assigned to available nodes in a round-robin fashion. It is implied by the graph that the `IntStream`, `Vector`, and `Integer` objects are used privately by each `StreamProcessor`, they therefore do not need to be remotely invokable, or be considered by a consistency protocol if a DSM system is used. Under the assumption that the other objects of the program are all assigned to a single node, it can be inferred that actually only the `MinFinder` itself is ever invoked across a distribution boundary, and must thus be remotely invokable (see [Spi00b] for a more detailed discussion of possible distribution analyses).

5.3 Case Study 3: Chess Opening Database

Figure 6 shows the object graph of a graphical database for chess openings. The program has about 2,500 LOC; the object graph is computed in 52 seconds (detailed results in section 5.4). Unlike the previous examples, we have greatly simplified this graph for presentation.

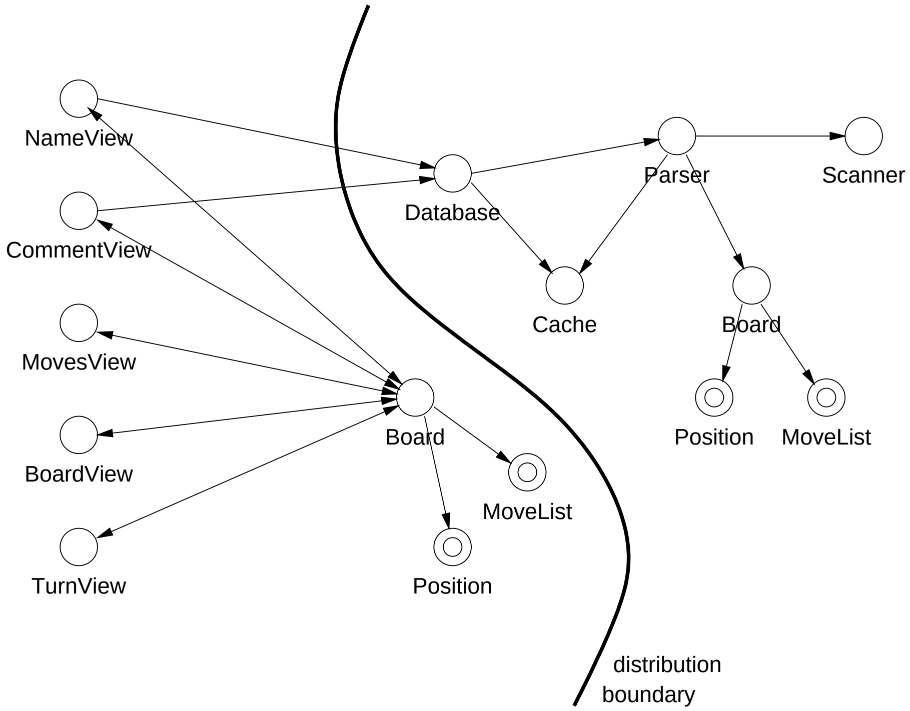


Figure 6: Object Graph for a Chess Opening Database

The program is intended to help chess players familiarize themselves with various openings. It displays a graphical chess board on which the user can make arbitrary moves; the program looks these moves up in a database and displays the name of the corresponding opening, and possibly a commentary on the move. The database is implemented as a simple text file.

The object graph shows the graphical user interface on the left; the objects that implement the database are on the right. There are actually two separate `Board` objects: one is used as the application model for the graphical chess board on the screen; the other is used internally by the `Parser` to interpret algebraic notation found in the text file. Some interesting properties can be shown regarding these two objects:

- The `Position` and `MoveList` objects used internally by the two `Board` objects are indefinite, because they are not created initially. However, despite the uncertainty about their actual numbers at run-time, the graph makes it clear that each `Board` has its own private objects of those types, and does not pass them to the outside.

- The left `Board` object communicates heavily with the user interface objects on the far left side. These interactions are realized through the Subject-Observer pattern, i.e. the `Board` object is a subclass of a `Subject` class that stores a list of `Observer` objects, which are updated on request. Of course, the *right* `Board` object also has such a list. However, reference propagation shows that none of the `Observer` objects is ever registered with the right `Board` object, and therefore it cannot invoke methods of them at run-time.

The information implied by the object graph allows for an efficient distribution of the program, turning it into a client/server application where the graphical user interface resides on a client machine, while the database is on the server. Using the distribution boundary shown in figure 6 (which can be found be graph partitioning), only the `Database` object needs to be remotely invokable, and a maximum of fast local communication is achieved (see [Spi00a] for a more detailed discussion of this example).

5.4 Performance

We have implemented the algorithm using the *Barat* framework for static analysis of Java programs [BS98]. The implementation itself comprises 2,000 lines of Java code (non-comment, non-blank).

Using this implementation, we have run the algorithm on a set of programs, ranging from 75 to 10,000 lines of code (excluding the Java standard library). The programs are briefly described in table 1. Each program was written by a different author; none was adapted for the analysis in any way. Experiments were made using JDK 1.2 (with JIT enabled) on a Sun UltraSparc 10 with 128 MB of memory, running under Solaris 2.6.

program	description
buffer	producer/consumer example
hamming	Hamming's Problem (Salishan benchmark)
rc5	RC5 cracking program
paraffins	Paraffins Problem (Salishan benchmark)
trace	simple ray tracer
sepia	graph drawing demo
chess	chess opening database
z3	Z3 machine simulator
jhotdraw	drawing application framework
vgj	graph drawing tool
javafig	Java version of xfig (presentation viewer)

Table 1: Example Programs

program	size		objects				edges		
	lines[a]	types[b]	static	concr.	indef.	total	creat.	ref.	usage
buffer	74	10	1	6	0	7	6	8	7
hamming	174	24	5	4	5	14	9	22	17
rc5	263	11	2	10	1	13	11	23	22
paraffins	556	43	7	1	52	60	53	296	205
trace	915	47	5	0	31	36	85	257	177
sepia	1,176	49	6	4	124	134	128	2,728	854
chess	2,474	133	21	37	58	116	98	446	275
z3	3,917	164	35	142	72	249	217	733	464
jhotdraw	6,163	276	39	22	250	311	272	9,236	2,848
vgj	10,352	197	47	15	319	381	375	7,633	4,700
javafig	10,699	218	49	25	385	459	445	4,263	3,844

[a]without comments and blank lines
[b]excluding standard library, except when directly referenced

Table 2: Analysis Results

program	step 1[a]	step 2	step 3	step 4	step 5	total time
buffer	2.6	0.4	0.5	0.0	0.0	3.5
hamming	4.6	0.5	0.5	0.1	0.0	5.7
rc5	2.8	0.6	0.8	0.2	0.0	4.4
paraffins	7.4	0.8	1.6	1.6	0.0	11.4
trace	10.5	1.0	2.8	1.2	0.0	15.5
sepia	9.1	1.2	4.5	39.5	0.1	54.4
chess	34.2	4.0	10.5	3.0	0.0	51.7
z3	71.0	6.6	6.6	9.7	0.3	94.2
jhotdraw	156.2	22.4	29.1	767.7	0.9	976.3
vgj	124.9	12.4	21.6	682.1	0.4	841.4
javafig	182.8	13.1	27.6	124.0	0.3	347.8

[a]includes I/O and parsing

Table 3: Times needed for Analysis (in seconds)

The results shown in tables 2 and 3 indicate that the algorithm generally scales well for programs up to 10,000 lines of code, with computation times on the order of several minutes at most. The number of reference and usage edges, and the times needed to compute them, is strikingly high for some programs, though, and might turn out problematic for even larger programs. We are tackling this problem in the following ways:

- When visualizing the object graphs, displaying several thousand reference or usage edges is clearly not useful. We have found it convenient to display only the creation edges at first, and to lay out the graphs according to these. This gives a very intuitive insight into the hierarchical structure of a program. Our analysis tool then lets the user selectively display reference or usage edges leading to or coming from a certain object in the graph. We have found this an excellent means to explore the run-time structure of a program.

- The high numbers of reference and usage edges are generally due to fine-grained objects which are extensively passed around and hence, aliased, within the object graph. For larger programs, it could be useful to either suppress such small objects from the graph completely (if no data passes through them), or to collapse the types of such objects into a single indefinite object. This could be done interactively by the programmer, but automatic criteria, e.g. depending on object size, are also conceivable.

6 Possible Improvements

In the form presented here, the algorithm has already proved useful for program understanding and distribution analysis. However, there is still room for improvements such as the following:

- To consider *exceptions* as full objects (see section 3.2) means to model the throwing of an exception as a data flow event; just as if a method had multiple return types. To realize this, the algorithm needs to annotate every method declaration with the types of exceptions it may throw, and to propagate these sets of exceptions up in a call graph, which may be constructed *ad hoc* using simple hierarchy analysis without much loss of precision. In a simple solution, an object automatically imports any indefinite exception objects owned by objects that it calls methods of; this approach does not consider whether the object actually *catches* all of these exceptions, or passes them on to its own callers. More sophisticated exception analysis, some of which could readily be incorporated into our algorithm, has been described in [RM99].

- When indefinite objects are added to the object population, they can only create further indefinite objects (see section 3.3). This may result in the loss of some precision that is actually still inherent in the algorithm. For example, in the program shown at the beginning in listing 1, if the `Worker` objects were not allocated initially by the `Main` object, the object graph would be folded and contain only an indefinite `Worker` object and an indefinite `Algorithm` object, thus losing the information that there is a one-to-one correspondence between these objects. One way to remedy this would be to mark reference edges as *one-to-one-edges* initially, and remove this property if edges are exported or imported. A more general solution would be not to consider individual objects as indefinite, but rather to introduce *summary subgraphs*, which may contain concrete objects, and yet be considered indefinite as a whole.

- The algorithm could be combined with flow-sensitive techniques to provide additional precision, while still capitalizing on the object-oriented perspective we introduced.

7 Conclusions

We have shown that existing approaches to static analysis of object-oriented programs are mostly concerned with type-level information. While this is sufficient for common compiler optimizations, we are currently seeing the advent of other kinds of high-level analyses such as distribution analysis and concurrency analysis. These require instance-level information, i.e. approximation of object graphs, but little has yet been done to tackle this problem. The algorithm that we presented here is a step towards filling this gap. Unlike some previous work, our algorithm embodies a decidedly *object-oriented* view of the problem, which enables high accuracy even though the algorithm is *flow-insensitive*, which in turn means that it has low computational complexity.

To our knowledge, our algorithm is the only true object graph algorithm that has fully been implemented for a main stream language (Java), and validated on a range of non-trivial example programs of considerable size. We have found the resulting object graphs highly descriptive in terms of program understanding, and we are now using the algorithm as part of a larger system named *Pangaea*, which will be able to *distribute* centralized Java programs automatically [Spi00a, Spi00b].

References

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[BS98] Boris Bokowski and André Spiegel. Barat – A front-end for Java. Technical Report TR B-98-09, Freie Universität Berlin, December 1998.

[Cor98] James C. Corbett. Constructing compact models of concurrent Java programs. In *Proc. ACM SIGSOFT Symposium on Software Testing and Analysis*, pages 1–10, 1998.

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proc. 26th Symposium on Principles of Programming Languages, POPL '99*. ACM, January 1999.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. Programming Language Design and Implementation, PLDI '90*, pages 296–310. ACM, June 1990.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proc. OOPSLA '97*. ACM, 1997.

[GI98] Joseph Gil and Alon Itai. The complexity of type analysis of object oriented programs. In *Proc. ECOOP '98*, number 1445 in LNCS, pages 601–634. Springer, July 1998.

[Hau98] Bernhard Haumacher. Lokalitätsoptimierung durch statische Typanalyse in JavaParty. Master's thesis, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, January 1998.

[PC94] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proc. OOPSLA '94*, pages 324–340. ACM, October 1994.

[PH98] Michael Philippsen and Bernhard Haumacher. Locality optimization in JavaParty by means of static type analysis. In *Proc. 7th International Workshop on Compilers for Parallel Computers CPC '98*, pages 34–41, Linköping, June 1998.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA '91*, pages 146–161. ACM, 1991.

[RM99] Martin P. Robillard and Gail C. Murphy. Analyzing exception flow in Java programs. Technical Report TR-99-02, University of British Columbia, March 1999. Submitted for publication.

[SH97] Marc Shapiro, II and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proc. 24th Symposium on Principles of Programming Languages, POPL '97*, pages 1–14, Paris, France, January 1997. ACM.

[Spi00a] André Spiegel. Automatic distribution in Pangaea. In *Proc. Workshop on Communications-Based Systems, CBS 2000*, April 2000.

[Spi00b] André Spiegel. Efficient distribution by static analysis. Technical Report TR B-00-12, Freie Universität Berlin, FB Mathematik und Informatik, June 2000.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proc. 23rd Symposium on Principles of Programming Languages, POPL '96*, New York, January 1996. ACM.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proc. 23rd Symposium on Principles of Programming Languages, POPL '96*, pages 32–41. ACM, January 1996.

[VHU92] Jan Vitek, R. Nigel Horspool, and James S. Uhl. Compile-time analysis of object-oriented programs. In *Proc. CC '92, 4th Int. Conf. on Compiler Construction*, number 641 in LNCS, Paderborn, Germany, 1992. Springer-Verlag.

Appendix A: Producer/Consumer Example

The listing shown below is the complete code of the producer/consumer example (shown as "buffer" in section 5.4). The code was taken from [Cor98] without modifications, except that `Integer` objects were used instead of primitive integers.

```
public class IntBuffer {

  protected Integer[] data;
  protected int count = 0;
  protected int front = 0;

  public IntBuffer (int capacity) {
    data = new Integer[capacity];
  }
  public void put (Integer x) {
    synchronized (this) {
      while (count == data.length)
        try {
          wait();
        } catch (Exception e) {
          e.printStackTrace();
        }
      data[(front + count) % data.length] = x;
      count = count + 1;
      if (count == 1)
        notifyAll();
    }
  }
  public Integer get() {
    synchronized (this) {
      while (count == 0)
        try {
          wait();
        } catch (Exception e) {
          e.printStackTrace();
        }
      Integer x = data[front];
      front = (front + 1) % data.length;
      count = count - 1;
      if (count == data.length - 1)
        notifyAll();
      return x;
    }
  }
}
```

```
public class Producer implements Runnable {

  protected int next = 0;
  protected IntBuffer buf;

  public Producer (IntBuffer b) {
    buf = b;
  }
  public void run() {
    while (true) {
      System.out.println ("Put " + next);
      buf.put (new Integer (next++));
    }
  }
}

public class Consumer implements Runnable {

  protected IntBuffer buf;

  public Consumer (IntBuffer b) {
    buf = b;
  }
  public void run() {
    while (true) {
      Integer x = buf.get();
      System.out.println ("Get " + x.intValue());
    }
  }
}

public class Main {

  public static void main (String argv[]) {
    IntBuffer buf = new IntBuffer(2);
    new Thread (new Producer (buf)).start();
    new Thread (new Consumer (buf)).start();
  }

}
```

Listing: Producer/Consumer Example