Chapter 3

Pangaea

Pangaea is a system that can distribute centralized Java programs, based on static source code analysis and using arbitrary distribution platforms as a backend. It thus distributes programs *automatically*, in the following four senses:

- Pangaea automatically *analyzes* the source code of a program, estimating its run-time structure and determining distribution-relevant properties.
- Based on this information, the *distribution strategy* for the program can be decided upon. This involves where to place objects, and when to employ object migration. To specify the distribution strategy, the programmer is provided with a graphical view of the program, in which he may configure the program for distributed execution, assisted by automatic tools.
- After the program has been configured, the desired distribution strategy is *implemented* automatically by regenerating the program's source code for a chosen distribution platform.
- At run-time, the placement of some or all of the program's objects can be adjusted automatically based on their communication behavior. This is realized by an *asynchronous migration facility* supplied by Pangaea's run-time system.

The architecture of the system is shown in fig. 3.1. Pangaea can be considered a *distributing compiler*: The source language of this compiler is pure Java, the target language is the particular Java dialect (and possibly, interface definition or configuration language) of the distribution platform to be used. Like a classical compiler, Pangaea consists of a *front-end* (the static analyzer plus the user interface and configuration utilities), and a code-generating *back-end*. The back-end architecture is generic, so that code generators for arbitrary distribution platforms can be easily developed and plugged in.

By its nature, Pangaea assumes Java's native programming model of *Threads & Objects* (see page 14), and is geared towards *explicit* distribution platforms using remote invocation as their basic communication mechanism (page 15). This is because only under the explicit approach, aspects of the distribution are still visible in the source code, which is both a drawback of this model which Pangaea seeks to overcome, and a potential for optimization that Pangaea exploits.

In particular, Pangaea enables two kinds of optimization that are essential for platforms that follow the remote invocation model, and which can *only* be found by static analysis (as opposed to a clever run-time system):

- The first optimization is to find the *dynamic scope* of object references, and thus to determine which objects do communicate at run-time, and which objects can be statically

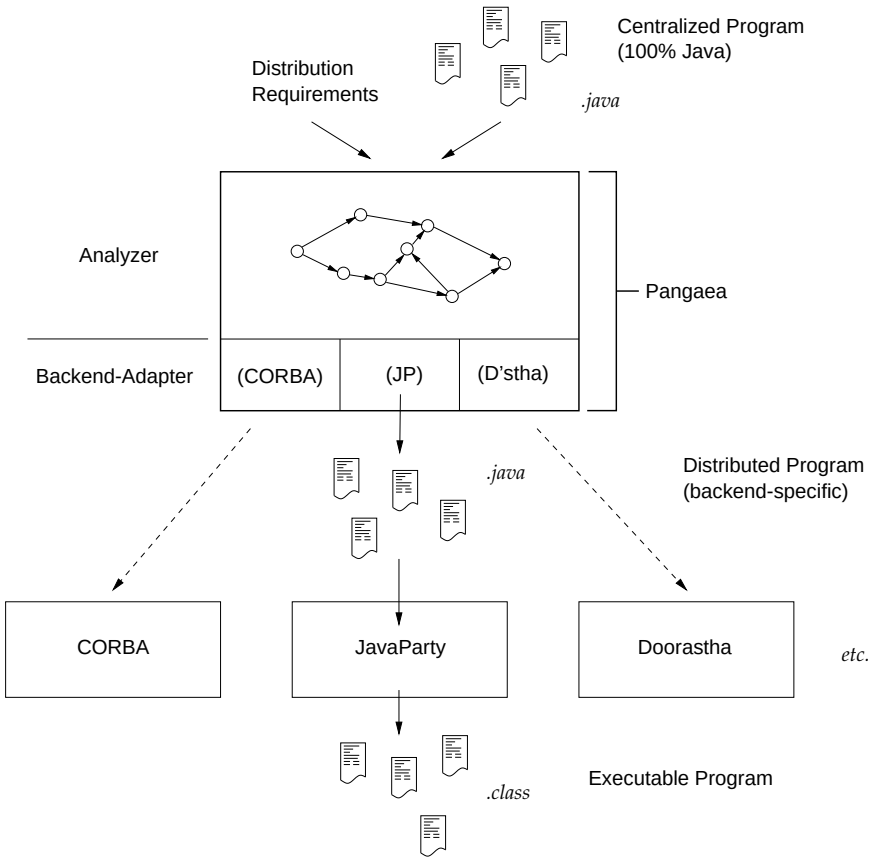


Figure 3.1: The Architecture of Pangaea

guaranteed *not* to communicate. Under a given distributed configuration, this information can be used to determine which objects must be remotely invokable, and which can stay local.

- The second optimization is to detect *immutable objects*, or objects that are used *immutably* within the program. Such objects can be replicated freely within the distribution environment, and passed by value in remote invocations without affecting the semantics.

According to the classification we gave in chapter 1, Pangaea targets both concurrent, computational programs (where the goal of distribution is speedup), and client/server programs (where the goal of distribution is to meet external constraints). In both of these areas, Pangaea adheres to the *single-program model*. In particular, this means that it cannot transform a centralized program into separate client and server applications, so that a server may be started independently and an arbitrary number of clients can connect to it. While it is conceivable that Pangaea could be extended to allow this kind of scenario, it is not an immediate focus of our work.

Pangaea also does not address the problem of *partial failure* (see also page 52). We consider the entire distributed system to be fault-free. In the event of a failure, we regard it as the distribution platform's responsibility to shut down the application in an orderly way. As we have argued in the previous chapter, this assumption is reasonable for the domain of concurrent, computational programs, while it is clearly not sufficient for the client/server world. In this domain, however, we maintain that a layer *below* the application logic would be required to solve the problem, which is beyond the scope of our work.

In the remainder of this chapter, we will describe the individual components of the Pangaea system in turn:

- We have developed a new *source code analysis algorithm* for Pangaea, which provides an object-oriented, macroscopic view of the program's run-time structure that is suitable for further distribution analysis. Unlike other analysis algorithms that have been published in the literature, our algorithm follows a decidedly object-oriented approach. We will describe this algorithm in section 3.1.

- Pangaea's *graphical user interface*, essentially a graph editor, provides a useful tool for program understanding in its own right, and is the means by which the programmer can specify distribution requirements. We will describe it in section 3.2.

- The act of specifying a distribution strategy for a program is called *configuring* the program. This involves where to place individual objects, which objects may be kept local on each machine, and for which objects asynchronous object migration should be employed. The programmer specifies the configuration in Pangaea's graphical user interface, assisted by automatic tools. We cover this process in section 3.3.

- After the configuration is complete, the types of the program are *classified*, which means, for example, that some types must be remotely invokable, others migratable, etc. The classification of the program's types is an abstract way of describing the implementation of the chosen distribution strategy. Classification is carried out automatically by Pangaea and forms the basis for the subsequent code generation step. We will describe the classification algorithms in section 3.4.

- Pangaea's *back-end code generators* regenerate the source code of the program, transforming it into a distributed program for the chosen distribution platform, while preserving the centralized program's semantics. This includes making some classes remotely invokable, turning some object allocation statements into remote allocations, and other transformations. There are back-end code generators for CORBA, JavaParty, and Doorastha. We will describe them in section 3.5.

- Pangaea provides its own *run-time system* in addition to that of the distribution platform. Its main purpose is to implement *asynchronous object migration*, which means that objects are monitored at run-time and moved to the partition from which they are most frequently accessed. Another purpose of the run-time system is to provide a general numbering scheme of partitions (machines) on each platform. We will describe the run-time system, particularly the migration subsystem, in section 3.6.

- Controlling a distribution platform in order to make it ready for program execution, and shutting it down properly afterwards, works different for each platform and can be very complex. Pangaea therefore provides a small *Launcher* utility that hides these differences and complexities, so that executing a distributed program becomes very similar to the centralized case. The Launcher is described in section 3.7.

3.1 Object Graph Analysis

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 the distribution analysis carried out in Pangaea; 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 covers the entire Java language except class loader interaction and reflection. It is flow-insensitive but context-sensitive, and therefore has a low computational complexity.

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 (Palsberg and Schwartzbach 1991, Plevyak and Chien 1994); where does a certain pointer variable point to at run-time (Steensgaard 1996, Shapiro and Horwitz 1997, Chatterjee et al. 1999); what method will actually be called in a dynamically dispatched call (Grove et al. 1997). 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 actually 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.

The algorithm described in this section analyzes the source code of a Java program to derive an object graph from it, representing the program's run-time structure. Despite its immediate use in Pangaea, 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) (Corbett 1998).

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 details.

According to the way in which analysis algorithms are usually classified (e.g. Shapiro and Horwitz (1997)), our algorithm is largely *flow-insensitive* (not considering the flow of control within methods), it is however *context-sensitive* in 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). Our initial experience shows that non-trivial real-world programs can thus be analyzed in acceptable time.

Before we describe the algorithm, we will review some existing techniques for the analysis of object-oriented programs, showing that they are not sufficient for constructing general object graphs.

```
    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 (...) { ... }
    }
```

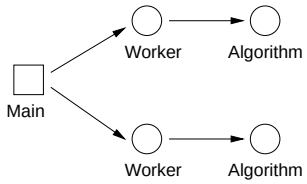


Figure 3.2: Example Program and Corresponding Object Graph

3.1.1 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 (Grove et al. (1997) 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 3.2, 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 (not shown in fig. 3.2), 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 3.2, 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 (Palsberg and Schwartzbach 1991, Plevyak and Chien 1994, Gil and Itai 1998) 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 (2000). 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 *Thread*s 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 Haumacher (1998)). 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 Chatterjee et al. (1999), 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 (Andersen 1994, Steensgaard 1996, Shapiro and Horwitz 1997, e.g.), 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.

Chatterjee et al. (1999) describe an approach called *Relevant Context Inference (RCI)*, 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 (Chase et al. 1990, Vitek et al. 1992, Sagiv et al. 1996, Corbett 1998). 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 (Chase et al. 1990, Sagiv et al. 1996).

More closely related to our work, Vitek et al. (1992) and Corbett (1998) 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.

3.1.2 Algorithm Description

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 3.3.

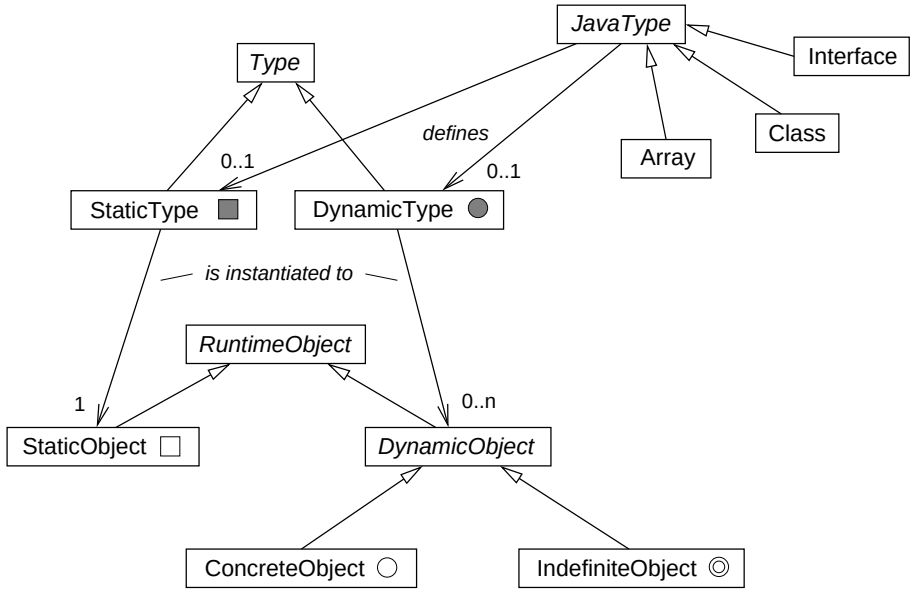


Figure 3.3: Ontology

In Java, the types of objects are called *reference types*, which can be classes, interfaces, or array types. 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 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

[1]There is, in fact, no special relation 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.

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 our description of step 3 below 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.

Step 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

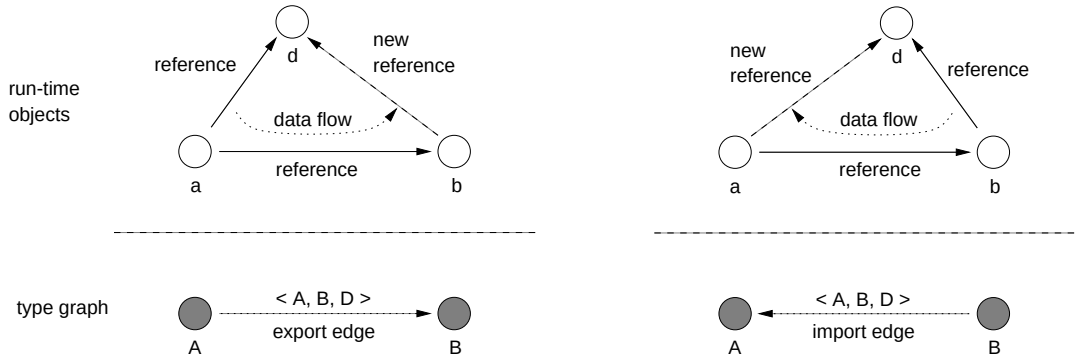
features poses a whole set of new problems for any static analysis algorithm. We are not adressing these in our work, and our algorithm cannot handle programs that make use of these features. At present, this does not seem like a serious limitation, as few programs actually fall into this category. Future 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.

Step 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 3.4: 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. 3.4): 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 restriction in principle; see the end of this section for a discussion of alternatives. Second, the above rules do need slight extensions for inner classes, which we omit here for brevity.

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

Step 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 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
  - the constructor[2] 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 in 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

[2]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 taken into account. As the details are trivial to spell out, we omit them here.

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.

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.

Step 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 fixed point is reached.

The actual algorithm corresponds exactly to the scenarios shown in fig. 3.4: 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 step 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.

Step 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 step 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.

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 we will discuss below, 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 fixed-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.

Case Studies

We will illustrate the kinds of results that our algorithm delivers in a two case studies now, looking in detail at the object graphs of two small to moderately sized programs. The performance of our implementation for several other programs will be given after that.

Case Study 1: Producer/Consumer

Figure 3.5 shows the program code of a simple producer/consumer program. This program is the same that Corbett analyzed in his paper (Corbett 1998), with one added complexity: rather

```
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();
  }

}
```

Figure 3.5: A Simple Producer/Consumer Program

than passing primitive integers from the producer to the consumer, we modified the program to use *Integer* objects, so that they would also be visible in the object graph, which is shown in figure 3.6.

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).

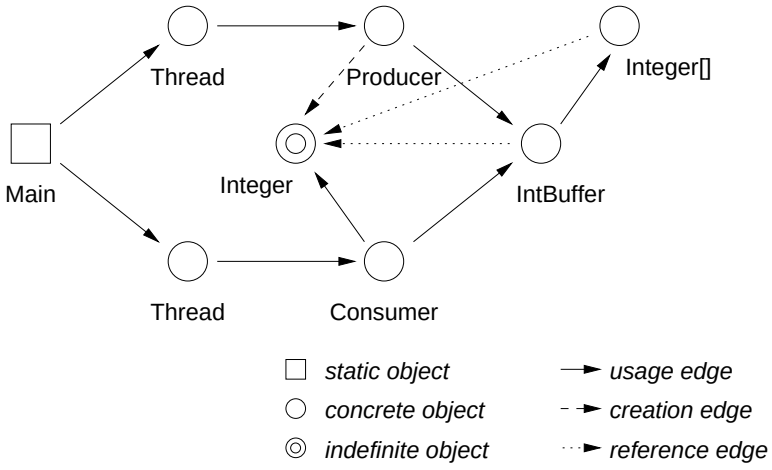


Figure 3.6: Object Graph of the Producer/Consumer Program

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 3.7 shows the object graph of a Java implementation of this algorithm[3]. The size of the program is about 170 LOC; the object graph is computed in 6 seconds (see the following section 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).

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 *StreamProcessor*s 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

[3]I would like to thank Michael Philippsen of the JavaParty team for giving me access to this program

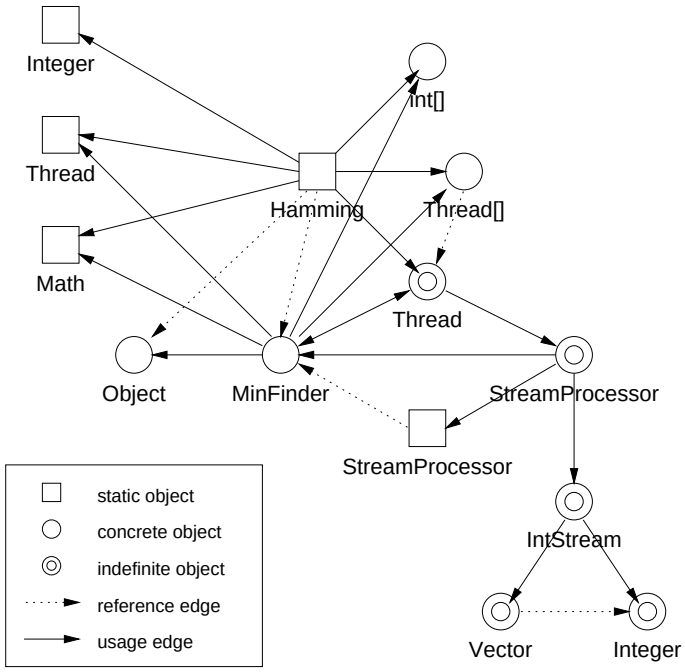


Figure 3.7: Object Graph for Hamming's Problem

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.

Performance

We have implemented the algorithm using the *Barat* framework for static analysis of Java programs (Bokowski and Spiegel 1998). 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 3.1. 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	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 3.2: Analysis Results

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 3.1: Example Programs

The results shown in tables 3.2 and 3.3 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

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 3.3: Times needed for Analysis (in seconds)

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.

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 3.1.2) 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 Robillard and Murphy (1999).

- When indefinite objects are added to the object population, they can only create further indefinite objects (see section 3.1.2). 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.

3.1.3 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 they form the basis on which the subsequent stages of Pangaea operate to distibute programs.

3.2 User Interface

Pangaea's user interface is a *graph editor* that allows the programmer to *view* the object graph produced by the analyzer, and *configure* the corresponding program for distributed execution by assigning objects to partitions. The configuration is usually done semi-automatically: the programmer can freely assign the objects, but he is being assisted by automatic algorithms that operate on the graph. Once the configuration has been completed, the user interface finally allows the programmer to trigger the generation step, which creates a distributed version of the original program, according to the configuration previously specified.

Figure 3.8 shows the main window of the graph editor[4].

To start the analysis, the programmer enters the name of the program's main class into a dialog. As described in the previous section, the "program" is defined as the *dependency closure* of this main class. For practical analyses, it is however essential to limit the scope of the analyzer, lest large parts of the Java standard library would be included in even the simplest programs. Screen output, for example, can be treated as an entirely local feature, and can therefore safely be excluded from the analysis. To control this, we have added a *type filter* to the analyzer, which can be adjusted in a separate dialog, shown in figure 3.9.

The type filter works by defining two special categories of types, *system types* and *irrelevant types*.

- A *system type* is a type that the analyzer captures, but only as a "black box", which means that its internals are not resolved. Objects of such types may therefore appear in the object graph, but the dependency closure does not extend beyond them.
- An *irrelevant type*, as the name implies, is completely irrelevant for the analysis, and not captured in the graph in any way.

The filters for these categories are implemented as a set of regular expressions to be matched against type names; these regular expressions can be edited in the type filter control dialog shown above. If an expression matches a given type name, Pangaea considers the type to belong to the corresponding category. Additionally, it is possible to set or clear a *subtype flag* for each

[4]The editor was written using the *JHotDraw* framework by Erich Gamma; available from `sourceforge.net/projects/jhotdraw`.

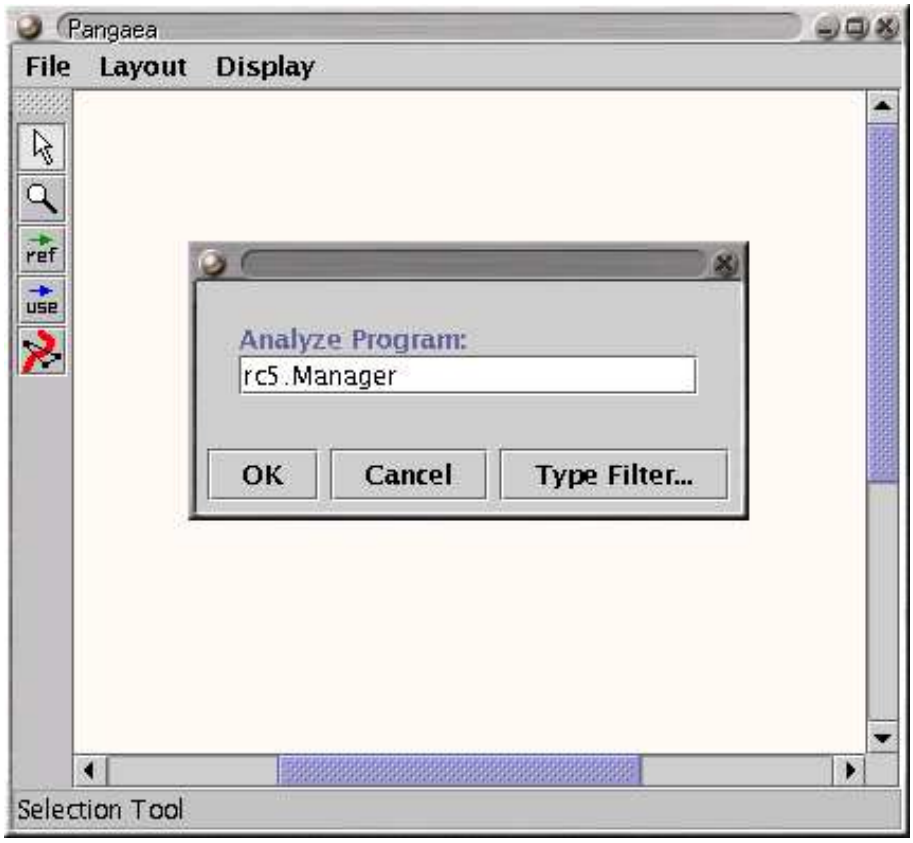


Figure 3.8: The User Interface of Pangaea

expression (shown as a checkbox in each row). When set, this flag means that any subtypes of types covered by an expression should also be blocked.

It is the responsibility of the programmer to ensure that the type filters do not block semantically relevant objects from the object graph. The default setting, shown in fig. 3.9, works fine for simple, screen-based programs, but needs to be adapted for more complicated cases.

After the analysis has completed, the main window shows the object graph of the program (fig. 3.10). Initially, the graph is layed out as a tree along its creation edges (this implies that each static object forms a separate tree, but normally only the tree rooted in the main class is relevant). We have found that this layout according to creation edges gives a very intuitive understanding of a program's structure, and is also simple to achieve algorithmically. Despite a lot of experimentation, we haven't found any satisfactory means of laying out the graphs according to their reference or usage edges. Because of their much more complicated structure, we also chose not to display these edges in the graph initially, although they can be activated from the *Display* submenu. Further options in this pull-down menu allow to mark immutable objects visually (by changing their labels to green), or to hide them from the graph completely.

On the left hand side, the programmer is provided with a number of tools to operate on the graph. From top to bottom:

Selection Tool The selection tool allows the programmer to select individual objects and drag them to a new position, to achieve a better optical layout of the graph. It is also possible to select and move several objects at once, either by dragging the pointer to create a "rubberband box", or by pressing the middle mouse button on an object, which selects the subtree rooted in that object. It is also possible to trigger the tree layout algorithm

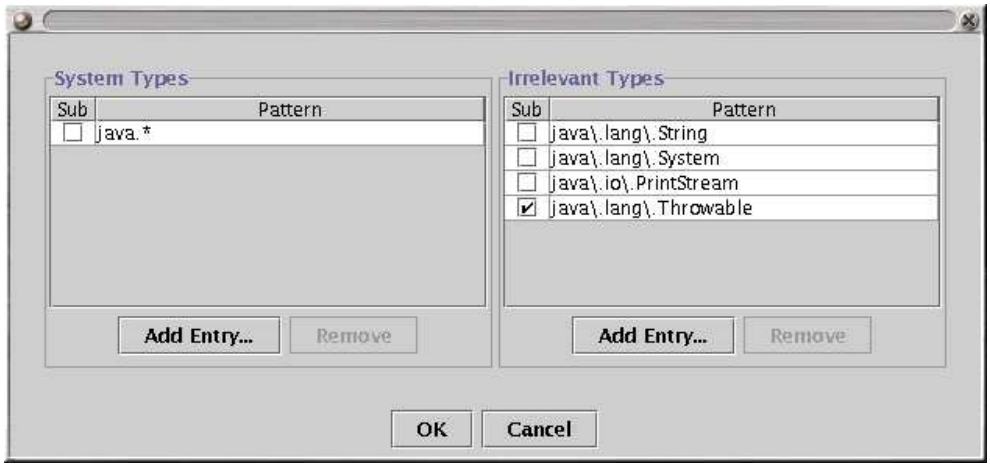


Figure 3.9: Type Filter Configuration Dialog

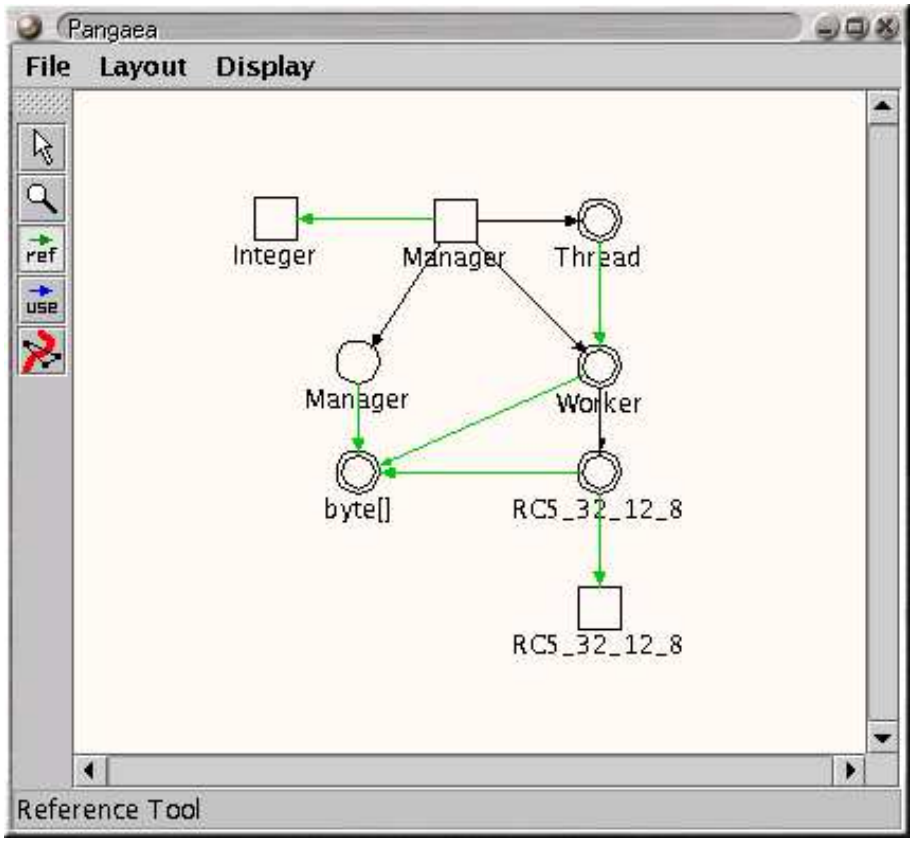


Figure 3.10: An Object Graph within the User Interface

for a subtree of objects, using a context menu on the right mouse button.

Zoom Tool The zoom tool lets the programmer zoom into and out of the graph, using a "rubberband box" to select an area to be displayed (left button), or by clicking the middle button anywhere in the display area, which centers the display at that location using a zoom scale of 1.0. Clicking the right mouse button reduces the zoom scale by a factor of two.

Reference Tool and Usage Tool The reference and usage tools allow selective display of reference and usage edges in the graph. When, for example, the reference tool is active, right-clicking an object shows all reference edges that point *to* that object, while a click with the middle button shows all reference edges that emanate *from* the object. (When a new object is clicked, any previously displayed reference edges are removed again, unless the shift key is pressed simultaneously.) We have found these two tools an excellent means to explore the reference and usage structure of a program, while it is usually impractical to display all reference and usage edges simultaneously, because of their sheer number.

Partition Tool The partition tool is used to configure the program for distributed execution. It allows the programmer to assign objects to partitions, to indicate whether objects should be migratable, and to trigger the automatic graph assignment algorithm. All of these actions are accessed via the right-button context menu that is activated by the partition tool (fig. 3.11). We will describe the precise meaning of these menu items in the following section.

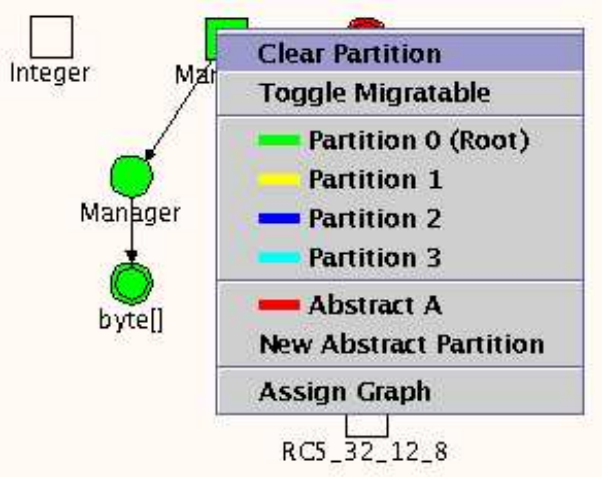


Figure 3.11: The Partition Menu

After the programmer has configured the program, using the tools and the automatic algorithms provided, the final step is to generate the distributed program. The dialog to initiate this, accessible from the *File* menu, is shown in figure 3.12.

In this dialog, the *Root path* is the directory below which the generated code should be written, and *From prefix* and *To prefix:* define a textual mapping for the program's type names: If a type's name starts with the *From prefix*, that prefix is replaced by *To prefix* in the generated program. This allows to write the generated code to a different package hierarchy than the original program.

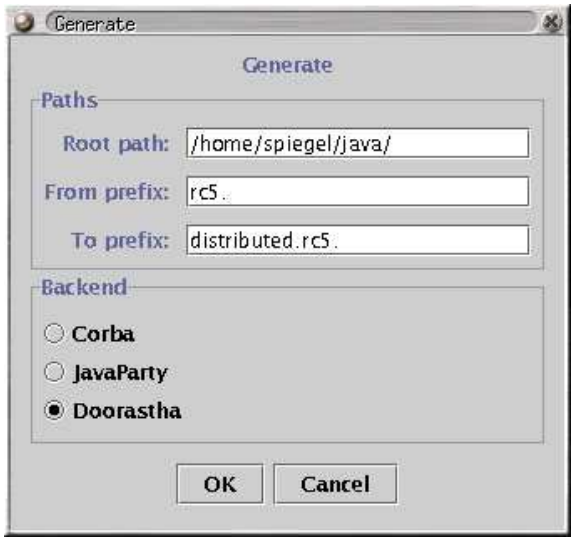


Figure 3.12: Generation Dialog

3.3 Configuration

To *configure* a program means to specify a *distribution strategy* for it. This includes where to *place* individual objects, but more than that: some objects may be kept local on each partition, or passed around by value, and thus have no particular location at all. Other objects may be under the control of the asynchronous migration subsystem, and so their location may vary over time. Yet others may be passed around *by move*, so their location varies synchronously.

The configuration thus includes all distribution-related aspects of a program, specified at the *object level*, and *independent from the particular distribution technology* that will later be used for the implementation.

When defined in this way, Pangaea is a good tool for configuring a program because (a) it has information about the program's object structure in the form of the object graph, and (b) its front-end/back-end architecture allows to abstract from the particular features of distribution platforms. Nonetheless, the above definition of a configuration is an ideal one and Pangaea only partially matches it. This is because (a) the object graph usually does not resolve all individual objects, but groups some of them together as indefinite objects. Thus, no individual distribution strategy can be specified for these objects. The second point (b) is that Pangaea's back-end adapters cannot provide an equal set of features for each platform. For example, there is no asynchronous migration subsystem for CORBA because that platform does not support object migration at all.

In the current implementation of Pangaea, an object can either be *assigned to a partition* (meaning that it should be located on that partition, at least initially), or it can be left *unassigned*. Additionally, an object can be marked as *migratable*; objects that are not marked in this way are *not migratable*. Altogether, this results in four possible combinations as shown in table 3.4.

An object that is assigned to a particular partition but is not migratable will simply reside on that partition, statically. If the object is migratable in addition, it will start out on its assigned partition, but can later move if decided by the migration subsystem. If an object is unassigned, this means that it will be a *local* object. A local object is created on the same partition as its creating object; if it is subsequently passed to another partition, it is passed by value (copied). When a static object is local, this means that a copy of it will be placed on each partition that

	migratable	not migratable
assigned	asynchronous migration	static location
unassigned	*not allowed*	local

Table 3.4: Possible Configurations for an Object

references it. Local objects thus have no particular location within the system at all. (It is clear that the fourth combination, *unassigned* and *migratable*, does not make sense and is therefore not allowed.)

The configuration does *not* specify whether an object is (or should be) remotely invokable. If an object is assigned to a particular partition, but is only referenced from within that partition, then it need not be remotely invokable. The classification of objects into global (remotely invokable) and local objects is therefore a derived property of the entire object graph, and will be dealt with in section 3.4. All that can be said at the configuration level is that unassigned (local) objects will never be remotely invokable, and that assigned objects *may* have to be remotely invokable.

Abstract Partitions

There is an important case often found in concurrent programming where it is insufficient to assign objects to particular partitions. In concurrent programs, there is often an indefinite number of *Thread* objects, usually created in loops controlled by configuration parameters. For example, a manager/worker application will usually have its number of workers controlled by a configuration switch. The resulting *Thread* objects, and all the objects created by these threads, will therefore be represented as *indefinite objects* in Pangaea's object graph. Hence, only the indefinite object that stands for *all* of the application's worker threads could be assigned to a partition, and thus all workers would end up on the same processor.

We have therefore added the concept of *abstract partitions* to Pangaea. An abstract partition is a partition that will, at run-time, be realized by multiple concrete partitions. To this end, Pangaea's code generator annotates the statement that creates a *Thread* object on an abstract partition with a call to Pangaea's round-robin scheduler, to obtain a partition number to use.

Abstract partitions do have an *identity*. (As concrete partitions are numbered 0, 1, 2, etc. in Pangaea, we label abstract partitions distinctively as partition A, B, C, etc.) This allows Pangaea to decide whether two objects are on the *same* abstract partition. At run-time, this means that individual instances of these (indefinite) objects will end up on the same concrete partition, and thus potentially do not need to be remotely invokable.

As an example, reconsider the graph of the *Hamming* program in figure 3.13 (we already discussed this graph in section 3.1, page 69). The "workers" in this program are the *StreamProcessor* objects shown in the lower right, along with their local *IntStream*, *Vector*, and *Integer* objects. When we assign the *Thread* objects to an abstract partition, and the hierarchy below the indefinite *StreamProcessor* objects to the same abstract partition, this means that these objects will in fact be local to their corresponding threads, placed onto the same processors, and do not need to be remotely invokable.

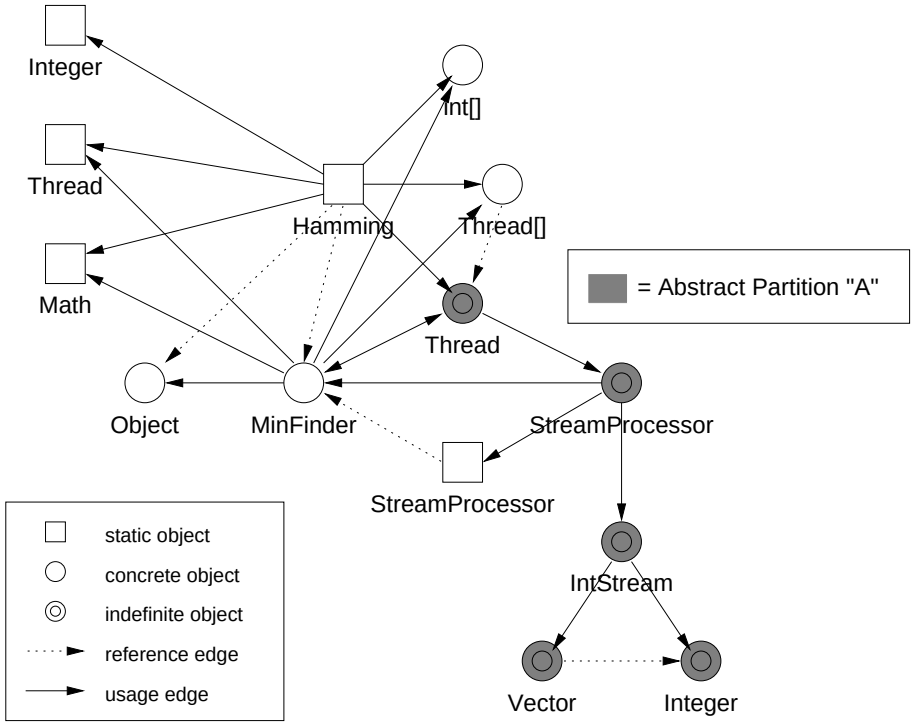


Figure 3.13: An Abstract Partition used for Hamming's Problem

Automatic Assistance

Although the process of finding and specifying a distribution strategy for a program is not fully automated in our current system, Pangaea does provide some automatic tools that assist the programmer in this task. In some cases, little or no additional work is required from the programmer, although he is still free to override or fine-tune any or all of the automatic decisions. Due to our focus on the other areas of automation (automatic analysis, automatic implementation, and automatic fine-tuning at run-time), we have barely scratched the surface of *automatic strategy-finding*. Other algorithms to enhance this process are easily conceived, given the existing Pangaea framework. Within this section, we will outline some of these other algorithms as well.

A prerequisite for any reasonable distribution strategy is to identify *immutable objects*, since these can always be local, and passed to other partitions by copying, without affecting the program's semantics.

An immutable object is an object whose state cannot, or does not change after it has been created. Depending on whether we use the words "cannot" or "does not", two definitions of immutability result. An object *cannot* change after it has been created if its type is *statically immutable*, meaning that it does not provide any modifying operations or direct access to object state. More formally, a type T is statically immutable if

- all of T's non-private instance fields (if it has any) are **final**,
- write accesses to private instance fields of T occur only in the constructors of T (or in field initializers, which is technically equivalent),
- only private fields of **this** instance are written to, and
- the supertype of T is immutable.

A weaker definition results if we use the words "does not change" above. Although its type may not be immutable, a particular object may be immutable *within a given program.* This is a definition at the level of individual objects, and may for example catch cases where a programmer has not set field protections as tightly as possible, or provided write methods simply out of habit, although they are never actually used.

Since Pangaea resolves the run-time structure of a program to the level of individual objects (not just the types), this property would be easily checkable by annotating the usage edges in the object graph as write edges vs. read-only edges. (At the type level, methods would have to be classified as write methods or read-only methods. Using this information, usage edges in the *type graph* could be classified as write edges or read-only edges; this could then be carried over into the object graph.) In our current system, however, we have only implemented a check at the type level because it was sufficient for the small number of programs we evaluated our prototype with. Pangaea's graph editor can mark immutable objects visually, or completely hide them from the object graph. The programmer can thus incorporate the immutability information in his decisions, but it is also used by the second automatic assistance tool, Pangaea's *graph assignment algorithm.*

It is a simple algorithm that helps distributing concurrent programs consisting of a number of threads, and objects used by these threads both internally and globally. The idea is simply to assign the static main object to the root partition, and *Thread* objects to other partitions in a round-robin fashion (the actual assignment may be deferred until run-time by using an abstract partition, see previous section). Every other object that is neither the main object nor a *Thread* object is assigned to the same partition as its *parent* object, where the parent is defined as either the creating object, or the executing *Thread* object in the case of a *Runnable* object. Objects of immutable types, however, are never assigned to a partition.

With this assignment scheme, the objects on which a thread operates privately are naturally co-located with that thread, while any objects used for global communication at least start out on a reasonable location which can later be adjusted by the run-time system. For examples of the use of the automatic assignment algorithm, see the RC5 and the Ray Tracer case studies in chapter 4. For these programs, the algorithm easily finds the correct distribution strategy and requires only very little additional programmer interaction to fine-tune it.

Another promising approach to automatic strategy-finding is *graph partitioning.* In client/server programs, there will usually be several "layers" of objects between those objects that *must* reside on the server and those that *must* reside on the client. An optimal distribution boundary between client and server is one that minimizes network communication, and it can be found by *partitioning* the object graph.

The general graph partitioning problem is to divide a graph in two or more parts, so that the number of edges crossing the boundaries is minimized. There are many variants of this problem, including for example the case where edges are weighted and the optimization criterion is to minimize the total weight of all boundary-crossing edges. Although the general graph partitioning problem is NP-hard (Lengauer 1990), there do exist good approximation algorithms that operate in linear time (Hendrickson and Leland 1993, Diekmann et al. 1994).

To use such an algorithm in Pangaea, it would be helpful (although perhaps not essential) to annotate usage edges within the object graph with their *estimated communication volume* at run-time. This could be done using similar heuristics as those in the *Orca* compiler (see page 41): if a call to an object occurs in a loop, it is taken to be executed "many" times; if it occurs in a branch of an if-statement, the estimated number of executions is reduced by 50%, and the like. The actual amount of *data* (parameters) transferred over a usage edge would not be a significant issue in most cases, since the order of magnitude of both local and remote calls is not influenced by the number or size of parameters unless they are *very* large (Spiegel 1998).

3.4 Classification

While the *configuration* of a program is a specification of a distribution strategy at the object level, independent of particular distribution technology, the *classification* of the program's types is the first step towards implementing the chosen strategy on a given platform. Classification is a transformation of the object-level strategy back to the type level, and it is partly influenced by the capabilities of the actual distribution platform to use.

Types are classified as *local*, *global*, or *migratable*, according to the following definitions and rules:

- Objects of *local types* are copied to each partition where they are needed, which means that they are passed by value in remote invocations, or, in the case of static types, replicated on each partition that references them. A type is classified as local if all of its instances in the object graph are *unassigned* (see page 77).

- Objects of *global types* are remotely invokable and, in the case of dynamic types, passed to remote partitions by network reference. The instance of a global *static* type is kept on a single partition, and accessed remotely if needed. A type is classified as *global* if

  - at least one of its instances in the object graph is assigned to a particular partition, and
  - at least one of its instances is created remotely or referenced remotely.

  An object is created or referenced remotely if at least one creating or referencing object is assigned to a different partition than the object itself. Note also that *usage* implies *reference* in our graphs.

- Objects of *migratable* types are monitored and potentially migrated at run-time by Pangaea's asynchronous migration subsystem. A type is classified as migratable if it is a global type according to the definition above, and additionally, at least one of its instances in the object graph is marked as "migratable".

- In addition to the above, immediate classification, *inheritance* needs to be taken into account. Normally, this means that if a type is global or migratable, then all of its supertypes and subtypes within the program need to have that property as well, although some distribution platforms allow exceptions to this rule.

Several notes are in order about this classification scheme. First and foremost, it is a rather crude and simple transformation from the object level back to the type level. This is witnessed by each occurence of the phrase "at least one of a type's instances" in the rules above. In effect, it means that we can lose much of the precision inherent in the object graph. For example, if only a single instance of a type is referenced remotely, it means that the entire type must be classified as global. An alternative would be to introduce a *cloning mechanism* at classification time: if different instances of a type are used differently within the graph (e.g. one is only used locally and another is used remotely), then the type can be cloned into a local type and a global type in the distributed program. This only works of course if the two instances are located in referentially disjoint areas of the object graph (although an automatic conversion mechanism for instances of cloned types could be conceived as well). We have not implemented a cloning mechanism in Pangaea, because for the relatively small programs that we distributed with our prototype, there was no actual need for it.

The types of the distributed program are not completely the same ones as those of the centralized program, however: during analysis, we split each type into a *static* and a *dynamic* type. If both of these types end up as being local in the distributed program, then we can *combine* them again into a single Java class. If however at least one of them is global, then they need to be kept separate in the distributed program, so two individual types with different names will be generated, one comprising the static part, the other the dynamic, non-static part.

Another important observation is that *arrays* can become global or migratable as well, since we treat them as ordinary reference types during analysis. At code-generation time, global arrays are wrapped into remotely invokable objects; we will describe this in more detail in the next section. It should be pointed out, however, that the lack of a cloning mechanism in Pangaea can have quite drastic effects with arrays: if a single integer array in an application needs to be remotely invokable, then this results in *all* of the application's integer arrays being wrapped into objects. (Although the price for this is not as high as one might believe, as we will see in one of our case studies in the following chapter.)

It is also a deliberate consequence of our rules that *local types* and *immutable types* are orthogonal concepts: a type can be local even if it is not immutable, and immutable types do not actually *need* to be local. As explained in the previous section, a type is then and only then classified as local if all of its instances are unassigned in the configuration. This gives the programmer freedom in the use of local (replicated) instances. For example, an I/O facility, even though it may not be technically immutable, can be replicated conveniently by leaving it unassigned in the object graph.

Finally, our implementation framework allows us to add backend-specific rules to the classification process. An example for this is a constraint in Doorastha that requires the types of instance variables of a globalizable type to be globalizable as well. The reason behind this is that the instance variables of a globalizable type *could* be accessed remotely, and therefore it is conservatively ensured that their types allow this. We are not particularly happy with this constraint, because Pangaea can do a better job at inferring whether these types do need to be globalizable. However, Pangaea's classification framework allows us to incorporate this rule for Doorastha as well, and identify these "secondary globalizable types".

3.5 Back-End Code Generators

Pangaea's back-end code generators implement the distribution strategy that is specified by the configuration and classification steps, by re-generating the program's source code as appropriate for the distribution platform to use. The resulting code, although human-readable, is not actually meant to allow further modification by the programmer. We regard the output as a kind of "distribution assembler" code that happens to be written in a high-level programming language.

In particular, Pangaea's code transformations involve:

- *Name Mapping* The program code is regenerated using a different package hierarchy, to avoid overwriting the original version. Additionally, the names of types that are separated into static and dynamic parts are changed to distinguish the types from each other. Since Java allows unlimited access between the static and dynamic parts of a type, all variables and methods are made public when types are separated. (In exotic cases, this could cause conflicts in method resolution, which would be solvable by systematic method and variable renaming.)

- *Migration Instrumentation* If a type has been classified as migratable, its code is transformed so that

- each instance gets an attached *watcher* object,
- the watcher object is notified each time a method of its observed object (the *base* object) is called,
- each of the base object's methods gets an additional parameter that identifies the partition from which it was called, and
- each call of a base object method in the program is transformed so that it supplies the local partition number in the additional parameter.

- *Globalization of Types* If a type has been classified as global (or migratable, which implies globality), it is transformed into a remotely invokable type, using the mechanisms of the chosen platform. This includes global arrays, which are wrapped into objects that are then made remotely invokable using the same mechanism as above.

- *Allocation Annotation* Each object allocation expression ("`new`" expression) for a global or migratable type is annotated with the partition number on which the object should be created, again using the mechanisms of the chosen distribution platform.

Of the above transformations, the first two are carried out generically, independent of the actual platform used; this is also true for part of the globalizing array transformation. The name mapping and migration transformations are straightforward, for some details of the latter, see Busch (2001). The globalizing array transformation is illustrated in the following example for an array of type *byte*:

```
public class Array$byte {

  private byte[] data;

  public Array$byte (int length) {
    data = new byte[length];
  }

  public Array$byte (byte arg0, byte arg1, ... byte arg7) {
    data = new byte[8];
    data[0] = arg0;
    data[1] = arg1;
     ...
    data[7] = arg7;
  }

  public void set (int index, byte value) {
    data[index] = value;
  }

  public byte get (int index) {
    return data[index];
  }

  public int length() {
    return data.length;
  }

}
```

Figure 3.14: Generic Code for a Global Array Object

The actual array is stored in the instance variable *data*. There are two constructors, one that takes only a length as an argument, and one that takes eight individual byte values. The former

is used for array allocations of the form *new byte[n]*, while constructors with individual argument values are generated on demand for each different kind of array initializer found in the program. Finally, there are *get()*, *set()* and *length()* methods that map the standard array operations. For completeness, additional methods would be required to map the C shortcut operators in expressions such as *a[i]++*, *--a[i]*, and *a[i] %= 5*. Careful attention would have to be given however to the intricate dependencies between these operators and the rules of evaluation order given in the Java language specification. For example, the following code:

```
int [] a = new a[1];
a[0] = 9;
a[0] += (a[0] = 3);
System.out.println (a[0]);
```

is required to produce a result of "12" in Java, while it is unspecified in ANSI/ISO C (§15.7.1 in Gosling et al. (2000)). A straightforward transformation of the above into method calls is bound to break the semantics, and a faithful implementation is likely to involve a lot more work, although we have no reason to doubt that it is in fact possible. Due to the prototypical nature of our system, we have chosen not to implement the shortcut operators, and instead slightly rewrite our example programs manually in the few cases where these operators were used.

In the remainder of this section, we will describe the code transformations for each individual platform, followed by a discussion of the limitations of our back-ends in section 3.5.4. To illustrate the transformations, we will use the smallest possible distributed program, a client/server version of *Hello World*. The centralized version's code is shown below:

```
public class Main {

  public static void main (String argv[]) {
    Server s = new Server();
    Client c = new Client(s);
    c.doIt();
  }

}
```

```
public class Client {

  private Server s;

  public Client (Server s) {
    this.s = s;
  }

  public void doIt() {
    System.out.println
                    (s.getMessage());
  }

}
```

```
public class Server {

  public String getMessage() {
    return "Hello, world!";
  }

}
```

Figure 3.15: "Distributed" Hello World, Centralized Version

The main method creates a *Client* and a *Server* object. On invocation of the client's *doIt()* method, it calls the server's *getMessage()* method, from which it receives the string "Hello, world!", which it then prints to its own standard output. The architecture is captured intuitively in Pangaea's object graph of the program, shown below, along with the distribution strategy we choose: the *Main* class, the *Client* object, and the *Server* object are assigned to partitions 0, 1, and 2, respectively.

3.5.1 CORBA

The CORBA adapter was developed in a separate Master's Thesis by Sven Knöfel (2000). For a general introduction to CORBA, please refer to our discussion in chapter 2, page 26. The basic idea of the adapter is to generate an IDL file which describes the global types of the program. From this IDL file, the IDL compiler supplied by the CORBA implementation generates the appropriate interfaces, stubs and skeletons. The classes of the centralized program are renamed

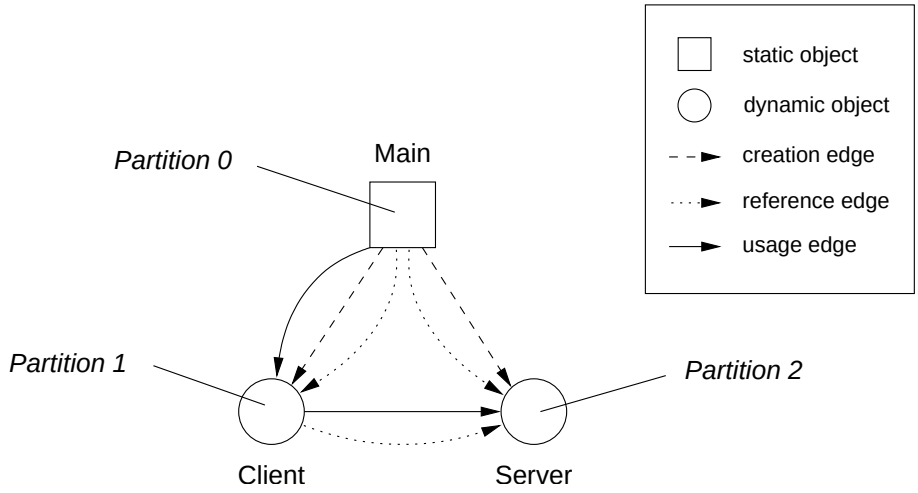


Figure 3.16: Object Graph of Distributed Hello World

and plugged into the inheritance hierarchy of the types generated by the IDL compiler. A special complication lies in the fact that Java is case-sensitive, while CORBA is not. To avoid name clashes, all Java identifiers are therefore transformed so that, for example, *RemoteObject* becomes *RemoteObject_0_6*, i.e. the positions of capital letters are appended to the identifier.

CORBA also does not have a concept of remote object creation. Since remote object creation is essential for our model of distributing programs, Pangaea's CORBA run-time system therefore supplies a generic *CorbaObjectServer* that is installed on each partition when the program starts up. The *CorbaObjectServer* allows the creation of arbitrary CORBA objects via remote invocations, and also supports transferring the byte code of the corresponding classes if they are not locally present. To allow the creation of objects with arbitrary constructor parameters, these parameters are transferred via generic sequences of CORBA *Any*s, and assembled into a constructor invocation via Java reflection within the *CorbaObjectServer*.

Figure 3.17 shows the IDL file and the *Main* class of the CORBA version of Distributed Hello World. The object creation expressions in the main method have been replaced with calls to newly generated, private methods that take both the constructor parameters and the number of the partition on which the object is to be created (*new$Server_0()* and *new$Client_0()*. Within these methods, the CORBA name server is contacted to look up the reference of the *CorbaObjectServer* on the target partition, which is then called to create the actual object. The second method, *new$Client_0()*, shows some additional work that is needed to wrap the reference to the *Server* object into a CORBA *Any* sequence and pass it to the object server.

By comparison, the transformed versions of the *Client* and *Server* class shown in figure 3.18 are relatively straightforward. The inheritance hierarchy has been changed and the classes have been renamed by adding an *Impl* suffix, since the interfaces that are used on the client side carry the original (case-sensitivity-mangled) names of *Client* and *Server*. The actual invocation of the *Server* in the *doIt_2()* method of the client shows no sign that the invocation is remote.

Although the code in the *Main* class is machine-generated and could be written somewhat more elegantly, it shows the amount of distribution-related work that is required by the programmer in typical CORBA applications, even to set up a very simple object structure as in our Distributed Hello World program.

Pangaea's CORBA adapter has several additional capabilities not demonstrated in this simple program. Among them is the ability to remotely create *Thread*s and *Runnable* objects, which is complicated by the fact that Java system types cannot be incorporated into IDL descriptions.

```
#include <CorbaObjectServer.idl>;

module distributed {
        interface Server_0 : pangaea::runtime::corba::RemoteObject_0_6 {
          wstring getMessage_3 ();
        };
        interface Client_0 : pangaea::runtime::corba::RemoteObject_0_6 {
          void doIt_2 ();
        };
};
```

```
package distributed;
import org.omg.CosNaming.*;
import org.omg.CORBA.Any;

public class Main_0 implements java.io.Serializable {
  public static void main (java.lang.String[] argv) {
    Server_0 s = new$Server_0 (2);
    Client_0 c = new$Client_0 (1, s);
    c.doIt_2 ();
  }
  private static Server_0 new$Server_0 (int parti$ion) {
    try {
      String args[] = {};
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      NamingContextExt nc = NamingContextExtHelper.narrow(orb.resolve_initial_references("NameService"));
      NameComponent [] name = new NameComponent[1];
      name[0] = new NameComponent("CorbaObjectServer"+parti$ion, "service");
      pangaea.runtime.corba.CorbaObjectServer os =
        pangaea.runtime.corba.CorbaObjectServerHelper.narrow(nc.resolve(name));
      Any[] parObjectName = new Any[0];
      Any[] parObject = new Any[0];
      Any re$ult = os.newInstance("distributed.hello.corba.Server_0",parObjectName,parObject);
      return Server_0Helper.narrow(re$ult.extract_Object());
    }
    catch ( Exception e ) {
      e.printStackTrace();
    }
    return null;
  }
  private static Client_0 new$Client_0 (int parti$ion, Server_0 s) {
    try {
      String args[] = {};
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      NamingContextExt nc = NamingContextExtHelper.narrow(orb.resolve_initial_references("NameService"));
      NameComponent [] name = new NameComponent[1];
      name[0] = new NameComponent("CorbaObjectServer"+parti$ion, "service");
      pangaea.runtime.corba.CorbaObjectServer os =
        pangaea.runtime.corba.CorbaObjectServerHelper.narrow(nc.resolve(name));
      Any[] parObjectName = new Any[1];
      Any[] parObject = new Any[1];
      parObjectName[0] = org.omg.CORBA.ORB.init().create_any();
      parObjectName[0].insert_string("distributed.hello.corba.Server_0");
      parObject[0] = org.omg.CORBA.ORB.init().create_any();
      parObject[0].insert_Object(s);
      Any re$ult = os.newInstance("distributed.hello.corba.Client_0",parObjectName,parObject);
      return Client_0Helper.narrow(re$ult.extract_Object());
    }
    catch ( Exception e ) {
      e.printStackTrace();
    }
    return null;
  }
}
```

Figure 3.17: IDL File and Main class of Distributed Hello World, CORBA Version

```
package distributed;

import org.omg.CosNaming.*;
import org.omg.CORBA.Any;

public class Client_0Impl
  extends pangaea.runtime.corba.RemoteObject_0_6Impl
  implements Client_0Operations {

  private Server_0 s;

  public Client_0Impl (Server_0 s) {
    super();
    this.s = s;
  }

  public void doIt_2 () {
    System.out.println(this.s.getMessage_3());
  }

}
```

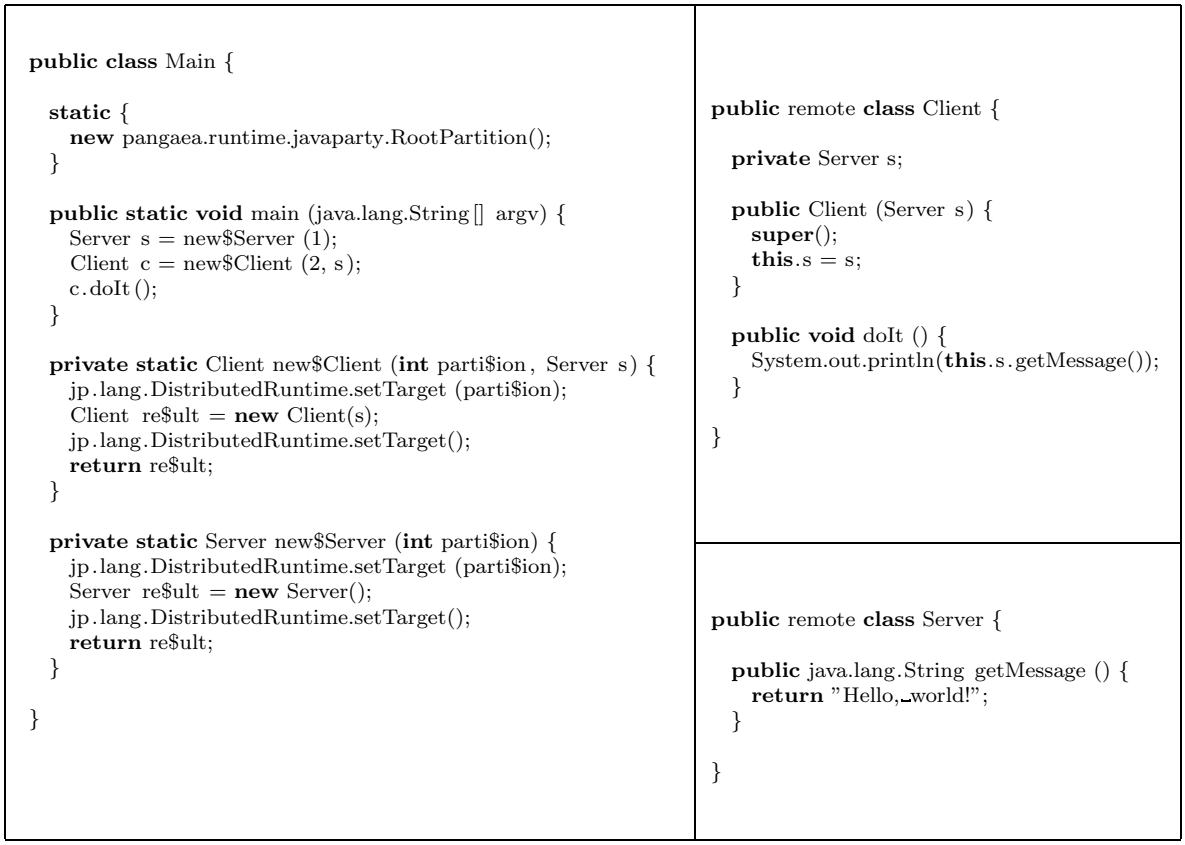
```
package distributed;

import org.omg.CosNaming.*;
import org.omg.CORBA.Any;

public class Server_0Impl
  extends pangaea.runtime.corba.RemoteObject_0_6Impl
  implements Server_0Operations {

  public java.lang.String getMessage_3 () {
    return "Hello,_world!";
  }

}
```

Figure 3.18: Server and Client class of Distributed Hello World, CORBA Version

The CORBA adapter also implements a protocol to pass objects by value using serialization, because it was developed at a time when objects-by-value where not part of the official CORBA specification. These features, along with several more complicated case studies, are described in detail in Knöfel (2000).

3.5.2 JavaParty

The JavaParty version of Distributed Hello World is shown in figure 3.19. The principle is the same as in the CORBA version, but the implementation is much simpler since JavaParty hides most of the complexity from the programmer (and likewise, from Pangaea's code generator). The *Client* and *Server* classes have been declared as "remote", which is the only change to these classes. As in the CORBA version, the object creations in the *Main* class have been moved into separate, private methods that allow arbitrary work to be done before and after the allocations. While a remote object creation shows no syntactic difference from a local one, JavaParty requires a separate call to the local machine's run-time system to set the target partition for the next creation statement, and to clear this setting again after the remote creation has been performed. This mechanism obviously creates a racing condition and is therefore suboptimal. JavaParty does have other mechanisms to specify creation targets, but none of them lent itself easily to our needs. With some additional effort, Pangaea could itself make sure that the object-creation methods are atomic, by introducing an additional per-machine synchronization variable.



```
public class Main {

  static {
    new pangaea.runtime.javaparty.RootPartition();
  }

  public static void main (java.lang.String[] argv) {
    Server s = new$Server (1);
    Client c = new$Client (2, s);
    c.doIt();
  }

  private static Client new$Client (int parti$ion, Server s) {
    jp.lang.DistributedRuntime.setTarget (parti$ion);
    Client re$ult = new Client(s);
    jp.lang.DistributedRuntime.setTarget();
    return re$ult;
  }

  private static Server new$Server (int parti$ion) {
    jp.lang.DistributedRuntime.setTarget (parti$ion);
    Server re$ult = new Server();
    jp.lang.DistributedRuntime.setTarget();
    return re$ult;
  }

}
```

```
public remote class Client {

  private Server s;

  public Client (Server s) {
    super();
    this.s = s;
  }

  public void doIt () {
    System.out.println(this.s.getMessage());
  }

}
```

```
public remote class Server {

  public java.lang.String getMessage () {
    return "Hello, world!";
  }

}
```

Figure 3.19: Distributed Hello World, JavaParty Version

Another change in the *Main* class is the addition of a static initializer that creates a Pangaea *RootPartition* object. This statement, which is executed at class-loading time even before the *main()* method is called, sets up Pangaea's run-time system that will be described in more detail in section 3.6.

For static objects that are assigned to a particular partition (including the static *Main* object in this example), a different assignment scheme than for remote object creations is used. JavaParty allows the programmer to specify a *ClassDistributor* class when the run-time environment is started. An instance of this class is called with the name of each "remote" class as it is loaded by JavaParty, and must return the partition number on which the class object (containing the static variables) should be allocated. Pangaea generates such a *ClassDistributor* class for each distributed program, containing the names of any global static types and the numbers of the partitions to which they have been assigned. Pangaea's *Launcher* automatically supplies the name of this *ClassDistributor* class to the JavaParty environment when it starts up.

We implement global arrays in JavaParty by using the generic code shown in figure 3.14, and simply adding the keyword *remote* to the class definition.

3.5.3 Doorastha

The Doorastha version of Distributed Hello World, shown in figure 3.20, is again very similar to the JavaParty version, except that tags in comments are used to indicate distribution-related properties. This means that the program could be translated by an ordinary Java compiler as well. The global *Client* and *Server* types are annotated with the *globalizable* tag, and the object creation statements receive the partition information from a *remotenew* tag. This has the advan-

```
public class Main {

  static {
    new pangaea.runtime.doorastha.RootPartition();
  }

  public static void main (java.lang.String[] argv) {
    Server s = new /*:remotenew :host=doorastha.RuntimeSystem.hosts[1]*/ Server();
    Client c = new /*:remotenew :host=doorastha.RuntimeSystem.hosts[2]*/ Client(s);
    c.doIt();
  }

}
```

```
public /*: globalizable*/ class Client {

  private Server s;

  public Client (Server s) {
    this.s = s;
  }

  public void doIt () {
    System.out.println(this.s.getMessage());
  }

}
```

```
public /*: globalizable*/ class Server {

  public String getMessage () {
    return "Hello, world!";
  }

}
```

Figure 3.20: Distributed Hello World, Doorastha Version

tage that no separate methods are needed for remote object creations, since they are preserved as being single expressions that can be used anywhere an expression is allowed. Doorastha does expect a host name instead of a partition number as the creation target; fortunately, the runtime system supplies the available host names in an array that is indexed according to Pangaea's partition numbers.

We implement global static objects by annotating each individual member with Doorastha's *unique* tag. A parameter of this tag specifies the host on which the member should be allocated; we can thus assign all members of a type to one particular host, as required by Pangaea's distribution model. Global arrays are implemented by taking the generic code shown in figure 3.14 and adding the *globalizable* tag.

3.5.4 Limitations

Pangaea's code generators are intended to preserve the semantics of the centralized program, possibly "correcting" any semantic changes that are introduced by the individual distribution platforms. For the most important case of this, the change in semantics for remote invocations with local parameter types (which are passed by value on all platforms), this is already ensured by Pangaea's configuration and classification mechanism. All types used as parameters in remote invocations are automatically classified as global because of the structure of the object graph, *unless* the programmer chooses to make them local by explicitly making the corresponding objects unassigned, *or* the assignment algorithm identifies them as immutable and leaves them

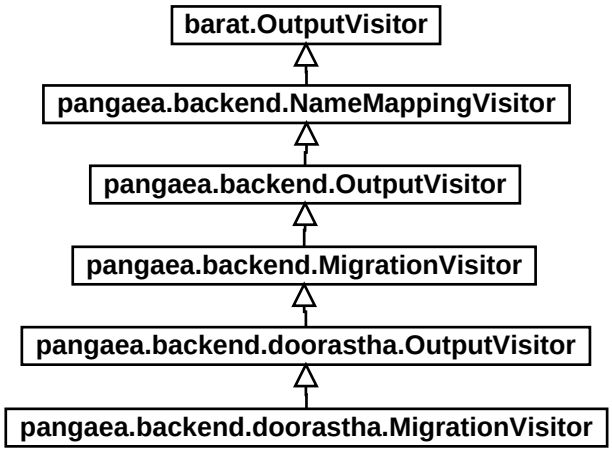


Figure 3.21: The Hierarchy of Pangaea's Output Visitors

unassigned by default.

There are however many more subtle changes such as the following:

- It is not possible to pass an instance of a Java system type to a "remote" JavaParty object by reference (since system types cannot be made "remote", see page 32). This could only be circumvented by an awkward trick involving an external static variable from which the "remote" object "pulls" the reference by itself.
- When a synchronized method of an object a calls an object b which in turn calls another synchronized method of a, this is possible in the centralized case because the executing thread owns the lock on object a. In a distributed setting where b is on another partition, the returning call will however normally be issued by another thread, and this thread will not be allowed to enter the locked object a. Thread identity would have to be managed on a global basis to solve this.
- Although immutable objects can be passed by value without semantic changes, object identity and, for example, hash code values are in fact affected by this. The corresponding operations would have to be reimplemented in a non-trivial way to solve this, or pass-by-value would have to be forbidden for types on which these operations are used in a program.

Pangaea, in its current form, cannot "correct" any of the above cases because of limitations in its code generation mechanism. *Barat*, the Java analysis framework with which Pangaea is implemented, does not allow the abstract syntax trees of a program to be modified (Bokowski and Spiegel 1998). The code generators are therefore implemented as a hierarchy of *Visitors* operating on the abstract syntax trees (Gamma et al. 1995). Each Visitor attempts to carry out a single transformation such as name mapping or instrumentation for the migration subsystem. The entire code generation has to be carried out in a single pass however, resulting in the need to combine all of the Visitors in a single inheritance hierarchy, as shown in figure 3.21.

It is clear from the picture that this mechanism has been stretched to its limits by the four main transformations that the current system implements (name mapping, asynchronous migration, globalization, and remote object creation). Additional, and more complex, transformations could only be carried out by modifying the abstract syntax trees directly, and incrementally. If Pangaea was implemented using an analysis framework that allowed this, each of the above cases could be handled individually, and more could be added as they are discovered.

This would never result in a *proof* that centralized execution semantics are completely preserved. However, neither is there a proof that any centralized execution environment fully complies with the specification of a real-world programming language. Sometimes, there is even reason to doubt that the specification itself guarantees some of its asserted properties (Drossopoulou and Eisenbach 1997). Compilers and execution environments are built *according* to specifications using careful up-front consideration and incremental improvement. Distributed execution environments must follow the same path, although it remains to be desired that distributed execution is incorporated into the specification from the beginning.

3.6 Run-Time System

The primary purpose of Pangaea's run-time system is to provide the application with *asynchronous object migration*. This means that at run-time, the placement of some objects may be readjusted based on observations of their actual communications behavior. We call this kind of migration *asynchronous* because it is not directly linked with the application's logic and control flow; it is performed by an entity that is external to the application itself. On the other hand, we use the term *synchronous object migration* for explicit migration commands within the application code, also in more structured forms such as pass-by-move parameters.

Technically, as we will see, asynchronous object migration might well be realized by code that is executed synchronously, i.e. within the same thread as the application code. Our term *asynchronous* simply means that the application code is not *aware* that migration analysis is carried out while it is executing.

Pangaea's migration subsystem was developed in a separate master's thesis by Miriam Busch (2001). The system is generic in two dimensions: (1) it can use arbitrary *strategies* for deciding when to migrate an object, and (2) it is independent of the actual way in which an object migration is *triggered* and performed on a given middleware platform.

The system does, however, assume that the underlying platform supports object migration. Among the platforms that we provide back-ends for, this is true for JavaParty and Doorastha, but not for CORBA. The run-time system, as we describe it here, has therefore only been implemented for the former two. (Pangaea's special run-time system for CORBA has already been described in section 3.5.1).

When the code for a migratable class is generated, Pangaea places hooks into it that will inform the run-time system of the communication behavior of the class's instances. Recall from section 3.5 that this involves:

- each migratable object gets an attached *watcher* object,
- the watcher object is notified each time a method of its observed object (the *base* object) is called,
- each of the base object's methods gets an additional parameter that identifies the partition of the caller, and
- each call of a base object method in the program is transformed so that it supplies the local partition number in the additional parameter.

At run-time, the base object interacts with the watcher and the rest of the migration subsystem as depicted in figure 3.22. For each call to the base object, the number of the calling partition is passed on to the watcher object, which in turn passes it to a *MigStrategy* object. At adjustable intervals (either after a fixed number of received calls, or a certain amount of

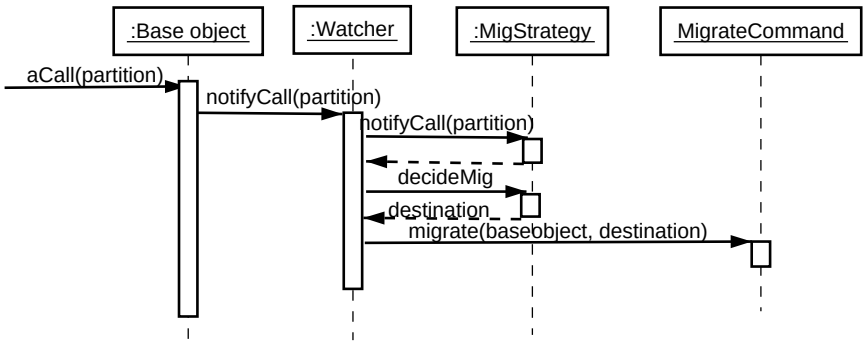


Figure 3.22: Interaction of Watcher and Strategy

time), the watcher asks the *MigStrategy* whether the base object should be migrated, and if so, where. If the *MigStrategy* advocates a migration, the watcher then calls the platform-specific *MigrateCommand* to actually perform it.

Each part of the above mechanism is configurable by using different implementations for the corresponding objects. This is illustrated by figure 3.23 which shows the hierarchy of the classes that make up the run-time system. There is a dedicated *Configuration* object that controls the choice of implementations for the *Watcher*, *MigStrategy*, and *MigrateCommand* interfaces (indicated by the grey associations in the figure).

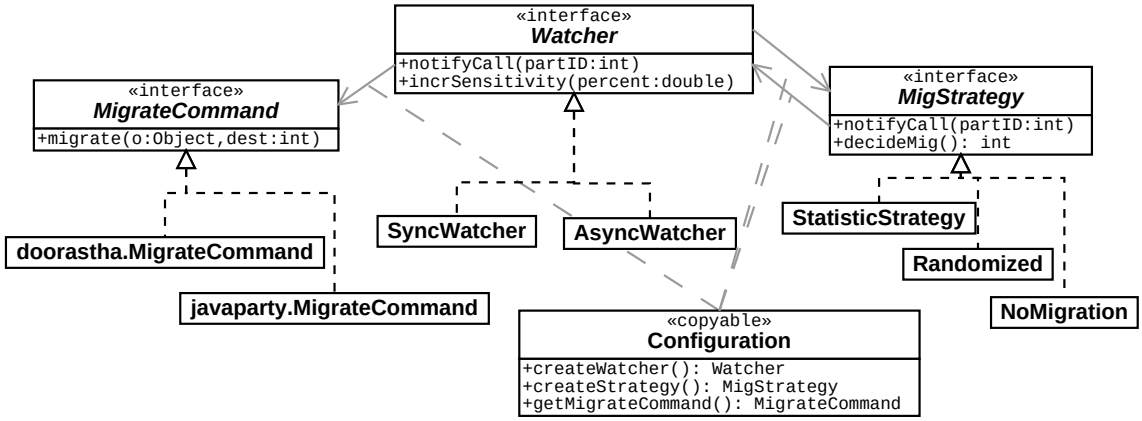


Figure 3.23: Class Hierarchy of the Run-Time System

Two implementations of the *Watcher* interface exist: the *SyncWatcher* asks the *MigStrategy* whether to migrate after a fixed number of calls received (the default is 50); the *AsyncWatcher* asks after a given amount of time has passed (the default is 100 milliseconds). Note that both *Watcher*s report each individual call synchronously to the strategy, they only differ in when they ask for a *decision* based on the accumulated information.

Three strategies have been prototypically implemented: the *StatisticStrategy* advocates a move to the partition from which the most calls were received, the *Randomized* strategy selects a destination at random, and *NoMigration*, as the name implies, never advocates a migration. Clearly, the *StatisticStrategy* is the most sensible one of the three, and in real use, it might be further subclassed to fine-tune its decisions. For example, a threshold might be imposed by which the number of calls from one partition must exceed those from other partitions for a migration to take place, etc.

The *MigrateCommand* encapsulates how a migration is actually triggered on the given platform. The abstract interface of it is:

```
public interface MigrateCommand {
  public void migrate (Object o, int destination);
}
```

meaning that the command takes the object that should be migrated, and the number of the destination partition as parameters. For JavaParty, this is implemented straightforwardly as:

```
public void migrate (Object o, int destination) {
  jp.lang.DistributedRuntime.migrate ((jp.lang.RemoteObject)o, destination);
}
```

which is to say that Pangaea's migration request is translated directly into an equivalent call to JavaParty's run-time system. (This further assumes that the partition numbers of JavaParty and Pangaea are equivalent, see below.) Doorastha, on the other hand, does not have such an equivalent migration command. In Doorastha, only the parameter mode pass-by-move is available to achieve object migration, and Pangaea therefore uses a trick: each partition gets a remotely accessible *Partition* object that resides on that partition and identifies it. To migrate an object to a partition, Pangaea calls a dummy method of the remote *Partition* object which has a single by-move parameter:

```
public void migrate(Object o, int destination){
  Partition.getPartition().getPartition(destination).take(o);
}
```

(The local partition, accessed via the static method call *Partition.getPartition()* offers a method *getPartition (destination)* that can return references to any other partition in the system. The dummy method *take()* is then called on the destination partition, which triggers migration via pass-by-move.)

The above mechanism has a subtle caveat, though. When the watcher object operates synchronously (as illustrated in figure 3.22), it implies that whenever a migration is triggered, there is an active method on the base object. Neither JavaParty nor Doorastha do however allow *strong* migration. With Doorastha, the situation is fortunately not as grave, since the migration does happen eventually, after the active method has returned. With JavaParty, however, migration simply fails. This leaves asynchronous watching as the only alternative on this platform. (We have not investigated any further how JavaParty's migration mechanism behaves when a method is called while a migration is in progress.)

An Exercise in Staticness: Setting up the Partition Objects

As we have seen, one prerequisite for the migration subsystem to work is to have a dedicated *Partition* object on each partition. Besides being used as a trick to trigger Doorastha's migration mechanism, these *Partition* objects serve a number of additional purposes:

- They introduce a consistent *numbering scheme* for partitions (starting with number 0, the "root partition" from which the program was started). While JavaParty does have a similar numbering scheme, there is no such thing in Doorastha (hosts are identified by their names).
- The numbering scheme itself is the prerequisite for Pangaea's simple *round-robin scheduler* that is used for the assignment of threads on abstract partitions (cp. section 3.3). It is the *Partition* object on the root partition that stores and gives out the master counter; all other *Partition* objects may obtain the next scheduled partition number from there via remote invocations.

- Finally, the *Partition* objects provide the migratable objects on each partition with access to the migration subsystem's *Configuration* object.

At start-up time, the run-time system must therefore ensure that there is a unique *Partition* object on each partition, and a special *RootPartition* object on the root partition (the class *RootPartition* naturally inherits from the *Partition* class). All *Partition* objects must have access to the *RootPartition*, and furthermore, all application objects that will subsequently be created on the partitions must have access to "their" local *Partition* object. This is illustrated in figure 3.24.

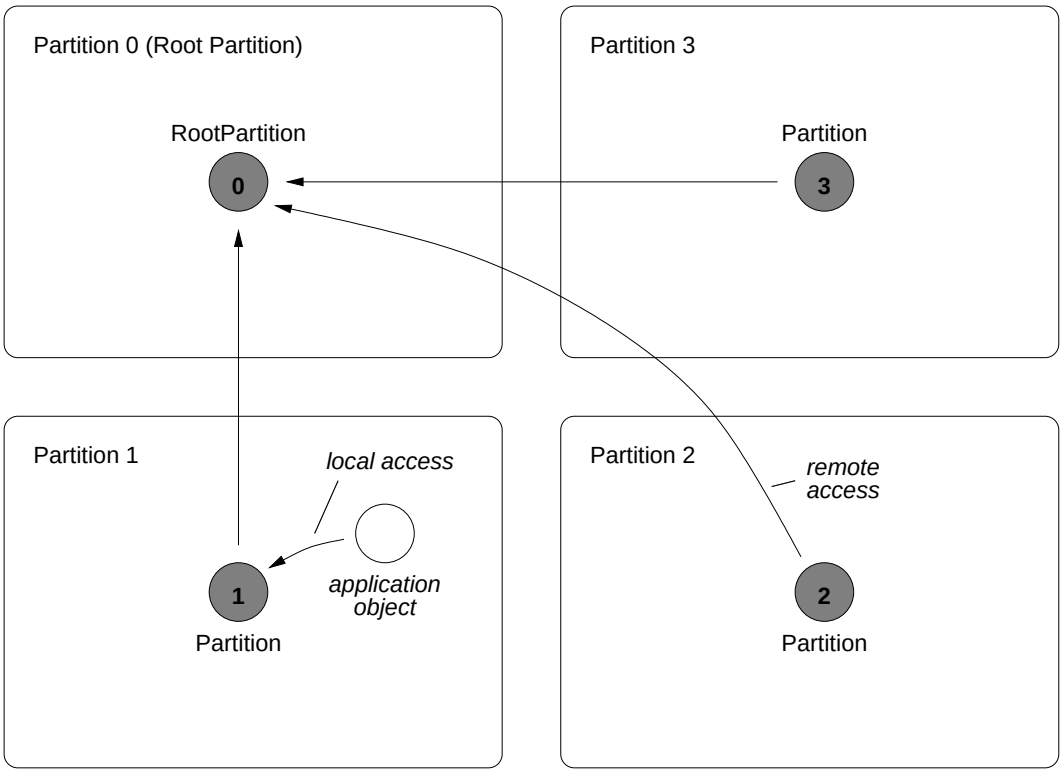


Figure 3.24: Objects Representing Partitions

Such a scheme is not entirely trivial to implement within the semantic framework of the Java language and the distribution platforms we use. The approach we have chosen is to create the *RootPartition* object first, and let it create the *Partition* objects on the other partitions remotely. The local point of access for each partition's *Partition* object is realized by storing its reference into a static variable. This however requires that the corresponding classes be *replicated* on each partition, including their static variables.

This scheme works well with Doorastha, because Doorastha by default replicates static variables on each host (a static variable must explicitly be marked as "unique" in order *not* to have it replicated). With Doorastha, it is therefore straightforward to create the *Partition* objects remotely and realize the local point of access as a static variable of the *Partition* class. (Each *Partition* instance, in its own constructor, stores a reference to *this* into the static variable.)

With JavaParty, the same scheme breaks however because (a) only "remote" (global) classes allow remote object creation, and (b) the static variables of global classes are always kept on a single host only. It is therefore not possible for the *Partition* objects to be created remotely *and* their class to function as the local point of access. It would be possible to give up on either of these requirements, and we choose the first one: an additional, global *PartitionLauncher*

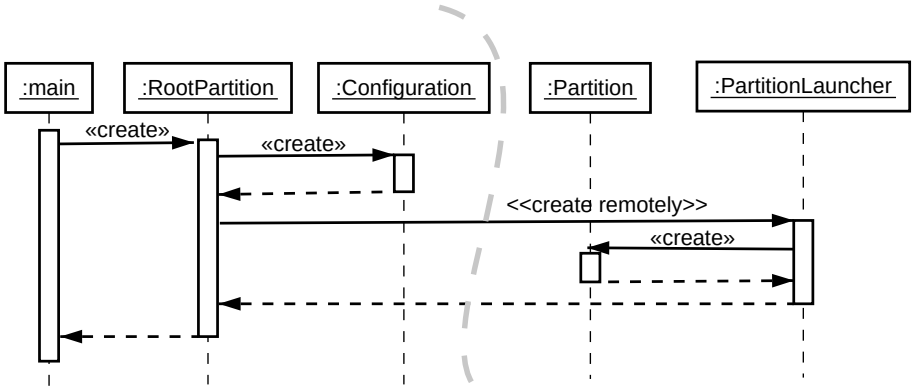


Figure 3.25: Initialization Mechanism for JavaParty

is introduced, which is itself being created remotely from the root partition and then creates the *Partition* object locally. The *Partition* class can therefore be local and function as the point of access. A consequence of this is however that *RootPartition* can no longer inherit from *Partition*, since "remote" classes cannot inherit from local classes in JavaParty. This results in the hierarchy of classes shown in figure 3.26.

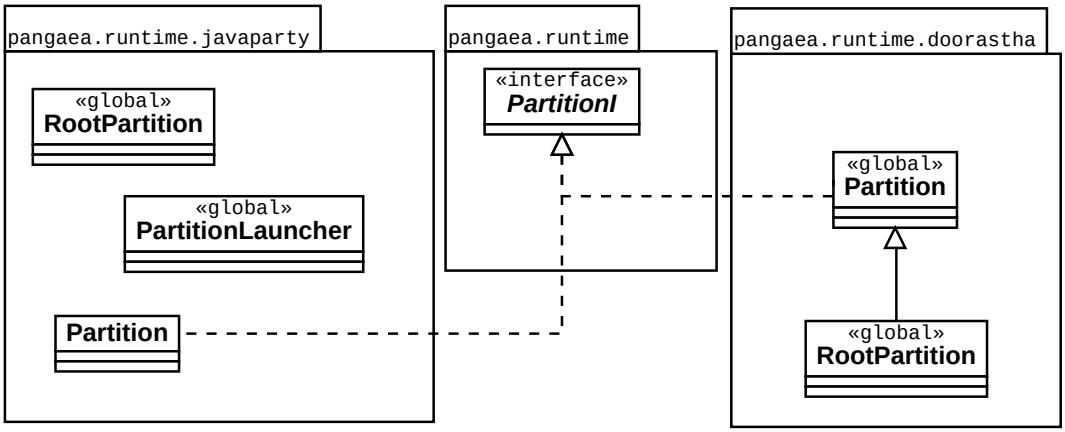


Figure 3.26: Implementations of the Partition Objects

Had we given up on the second requirement, and provided a point of access outside of the *Partition* class, then both *Partition* and *RootPartition* could have been "remote" and inheritance would have been possible. However, since local access to "remote" classes is not fully optimized in JavaParty (see chapter 4) and can be almost as slow as a remote access, we chose the former alternative although it breaks the inheritance hierarchy.

The difficulties thus encountered highlight the fact that the concept of *staticness* is a somewhat alien concept in the Java language itself, which in turn makes it difficult to adapt it to the distributed case.

3.7 Launcher

The distribution abstraction in Pangaea would not be complete if the user was finally exposed to the complexities of *starting* a program on one of the platforms supported. On each of them, this involves starting one or more server processes on each host, monitoring their output for diagnostic messages, then starting the program in a platform-dependent way, and finally shutting down the system in an orderly, and once again platform-dependent way.

Logically, however, the concept is simple: All the user wants to do is to run a program on a given set of hosts. Of course, we are saving a lot of complexity here because we adhere to the single-program model, and do not consider long-lived server components, as for example HERON does (page 21). Still, the single-program model is an important case where most of the platform-handling complexity is simply unnecessary.

We have therefore created a small *Launcher* utility that lets the user run a distributed program in almost the same way as a centralized Java program. The only additional information that needs to be supplied is the names of the machines to use. The Launcher is implemented as a small Java framework that is independent of the rest of Pangaea.

The command for starting a centralized program is

```
java package.MainClass arg1 arg2 ...  argN
```

The command for activating Pangaea's Launcher is the same, except that 'java' is replaced by 'pgjava' and the names of the hosts to use are inserted before the main class name:

```
pgjava host1 host2 ...  hostN package.MainClass arg1 arg2 ...  argN
```

The name of the main class (and hence, the end of the host list) is found by iterating through the argument list until the first name that represents a loadable Java class is found. Since in rare circumstances, this could be ambiguous (a host and a Java class could have the same name), a double hyphen may be inserted after the host list for disambiguation:

```
pgjava host1 host2 ...  hostN -- package.MainClass ...
```

The Launcher first determines what distribution platform to use for the program. To this end, Pangaea's code generator places a file named `PLATFORM` into the distributed program's main directory (the directory where the main class resides). This file contains a single line that identifies the platform for which the program was generated.

The Launcher then opens *ssh* connections to each host listed on the command line. Issuing shell commands on these connections, the Launcher first *starts* the platform, it then *executes* the program, and finally *shuts down* the platform again. Within the Launcher's implementation framework, these three steps are implemented individually for each platform. During the entire operation, the Launcher collects the output from each *ssh* connection and displays it on its own terminal, each line being prefixed with the name of the host from which it originated.

To *start* the platform means to create a distributed execution environment comprising all of the participating hosts, which is ready to accept programs for execution.

- For *CORBA*, this means to start a name server and place its object reference into a text file where the other components can find it. Then, Pangaea's *CorbaObjectServer* (cp. page 84) is started on each host.

- *JavaParty* requires a runtime manager *(jprm)* to be started on a dedicated host; once it is operational, individual JavaParty virtual machines *(jpvms)* can be started on each host.

- *Doorastha* does have a facility to start the platform in a centralized way, and to collect any output from the participating hosts in a centralized location (Doorastha's run-time system itself opens *ssh* connections to each host, for example). To make it fit with Pangaea's Launcher architecture, we actually use a simpler fallback mechanism where a *SpawnServer* is started on each machine. Java virtual machines are only created on demand during program execution, however if there is a *SpawnServer* running on a given host, it can be instructed to create the JVM itself and to collect any output from that JVM and write it through to its own controlling terminal.

Once the platform is operational, the program to run is executed. The Launcher in its current form does not do anything to *deploy* the code to the participating hosts, and neither is there any mechanism that partitions the application's code base into *components* (as in HERON, page 21). In Pangaea, we simply assume that there is a shared filesystem and each host has access to the entire application's byte code. This is not a restriction in principle, however. We simply haven't implemented any deployment mechanism because it was not necessary for the kinds of programs we studied. A common start, execute, and shutdown mechanism, however, was sorely needed. The Launcher would in fact be a good *abstraction* to integrate a deployment mechanism into as well.

Starting the program is simple on each of the platforms, given that the platform itself is already operational. It only takes a single command on each of them, the precise syntax of which is encapsulated by the Launcher.

After program execution, it needs to be ensured that the platform is shut down carefully and properly, because on each subsequent run, the Launcher will assume that the platform is *not* already started (see below for possible improvements on this).

- Shutting down the *CORBA* platform is implemented in a separate utility that is part of Pangaea's CORBA runtime system. It queries the name server to find the *CorbaObject-Server* on each machine, and then shuts it down by invoking a special method of it.

- *JavaParty* has a special utility, *jprk*, which encapsulates the shutdown mechanism. It only needs to be called once on one of the participating hosts.

- A two-step process is required on *Doorastha*: First, the JVM on each host needs to be terminated. This is accomplished by invoking a special *doorastha.Shutdown* utility on the root host, which transitively kills all JVMs on the other hosts. Second, all *SpawnServers* need to be terminated as well, which is done by invoking the *Shutdown* utility once more on each individual host.

In a more advanced version of the Launcher, it might be worthwhile to separate the start / execute / shutdown steps from each other, so that the platform could be brought up and used for an arbitrary number of program executions before it is finally shut down. The exact semantics of this would have to be worked out, however, in particular the question of whether already-loaded classes should be re-used in subsequent program executions, including the values of their static variables.