Automated Extraction of Abstract Object Models

by

Allison Leah Waingold

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2001 [June 2001]

Author ....

Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by ....

Daniel N. Jackson
Associate Professor
Thesis Supervisor

Accepted by ....

Arthur C. Smith
Chairman, Department Committee on Graduate Students

Automated Extraction of Abstract Object Models

by

Allison Leah Waingold

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Object modeling allows software designers and programmers to concisely describe the essence of a program's design, by describing program state. Automatically extracting object models from code can be useful for many reasons: to summarize the architecture of the program, to resolve anomalies in the design, or even to find bugs. This thesis describes Superwomble, a tool for automatically extracting object models from Java bytecode. The object model is generated using an unsound type analysis on the code. The unsoundness is an engineering tradeoff that in practice results in correct object models. The tool also allows for automatic generation of abstract object models, by applying user-defined abstraction rules to the models. The techniques for analysis and abstraction are described, and are evaluated on some sample programs.

Thesis Supervisor: Daniel N. Jackson
Title: Associate Professor

Acknowledgments

I am grateful to my advisor Daniel Jackson, for his technical guidance over the past several years. I also thank the other members of my research group, as many of them have also provided me with technical guidance throughout this project; thanks especially to Jon Whitney, who contributed to this thesis through his analysis work on the Java reimplementation of CTAS. Thanks also to the course staffs for 6.170 during the past two terms, many of whom have helped me to develop my understanding of software engineering topics. Thanks especially to Felix Klock, Rob Lee, and Jeremy Nimmer, who provided useful feedback on a variety of aspects of this project, especially with their suggestions for how to improve the Superwomble abstraction language. Finally, thanks to Robert O'Callahan for being a useful resource on all aspects of code analysis and for providing the low-level bytecode parser from his tool Ajax for Superwomble.

Contents

1 Introduction **13**

2 Object Model Syntax and Semantics **17**

3 Language Definition **21**

3.1 Language Semantics 21

3.2 Examples 25

4 Typing Rules **29**

4.1 Merging Types 33

4.2 Examples 37

4.3 Weaknesses of the Analysis 41

4.4 Implementation Notes 42

4.4.1 Method Dispatch 42

4.4.2 Approximations in the Control Flow Graph 44

4.4.3 Recursion 44

4.4.4 Approximating the Type Hierarchy 45

4.5 Extensions 45

4.5.1 Union Types 45

4.5.2 Dynamic Dispatch 46

5 Generating Object Models **49**

5.1 Object Model Annotations 52

5.2 Tool Features 53

5.3 Extensions 53

6 Abstraction 55

6.1 Examples 55

6.2 Example Abstraction Functions 58

6.3 Syntax and Interpretation of Abstraction Rules 59

6.4 Extensions 62

7 Results 71

7.1 Using Superwomble Models 71

7.2 Case Studies 72

7.2.1 Foliotracker 72

7.2.2 CTAS 73

7.2.3 Grappa 77

7.3 Analysis 77

7.3.1 Performance 77

7.3.2 “Correctness” of Results 78

8 Related Work 81

8.1 Related Type Analyses 81

8.1.1 Soft Typing 81

8.1.2 Ajax 82

8.2 Related Tools 82

8.2.1 Commercial Tools 82

8.2.2 Womble 83

8.2.3 Ajax 85

8.2.4 Other 87

9 Conclusions 89

List of Figures

2-1 r associates each A with n Bs and each B with m As. 18

2-2 The set of Bs associated by r with an A is fixed over the lifetime of an A. 18

2-3 A sample object model, showing the relationship between a company and its employees. 19

3-1 Operational Semantics of Language 23

3-1 Operational Semantics of Language 24

3-2 Bytecode for `Vector` example 27

4-1 Language of Type Expressions 29

4-2 Typing Rules (Primitive Instructions) 32

4-2 Typing Rules (Composite Rules) 33

4-3 Definitions of merge operations 34

4-4 A program fragment where unsound merging would affect Superwomble output. 36

4-5 Control flow graph for `Vector.addElement`. 39

5-1 (a) Object model of `Vector`. (b) Object model of `Company`, which uses `Vector`. 51

6-1 Object model of `Company` (a) without abstraction (b) with abstraction of `Hashtable`. 57

6-2 Object model of `Company` (a) without abstraction (b) with abstraction of `Vector`. 58

6-3 Syntax of Abstraction Language 60

6-4 Generic example of an object model (a) before abstraction, and (b) after abstraction. 61

6-5 Two possible representations for a graph ADT. 63

6-6 Superwomble-generated models for graph implementations. 64

6-7 Abstract graph models. 64

6-8 Abstract state of 3 graph models 65

6-9 Graph models after abstraction functions have been applied. 67

6-10 For each `Integer` id number, there is one `Person` in the employee relation of `Company`. 68

6-11 Superwomble-generated model of `Company`, with standard abstraction rules applied. 69

7-1 Superwomble-generated object model of Foliotracker. 73

7-2 Object model of Java reimplementation of CTAS. 76

7-3 Object model of Bridge pattern in Grappa. 77

8-1 Rational Rose-generated object model of Java reimplementation of CTAS. 83

8-2 Womble-generated object model of Java reimplementation of CTAS. . 84

8-3 Ajax object model of Java reimplementation of CTAS. 86

List of Tables

7.1 Runtime performance of Superwomble on sample programs 78

7.2 Frequency of unsound merges in Superwomble 79

Chapter 1

Introduction

Object modeling provides designers of object-oriented programs with a way to concisely documenting the essence of a program's design. We can represent the state of the heap during an object-oriented program's execution as a graph, where objects are nodes and field references are edges; this graph is called a *snapshot*, since it represents the heap state at one point in time. A *code object model* describes all possible legal snapshots; that is, all legal states of the heap. A node represents a class, and an edge represents a field reference. Edges are annotated with multiplicity information, which describes how many objects there are of each class in relation to each other; and mutability information, which describes how the relations between objects are allowed to change. [6]

This thesis discusses Superwomble, a tool to automatically extract object models. In some cases, a design-time object model was not created; in other cases, a designer may want to know whether the design-time object model is in sync with the actual implementation. In both of these cases, extracting an object model from code can be helpful for documentation, debugging, or overall program understanding.

Many existing tools extract some sort of object model from code. However, these tools are plagued by problems that make the tools difficult to use. In its simplest form, a tool could build an object model from its field declarations alone, and multiplicity could be determined from the presence of arrays. However, in a language like Java, which lacks parametric polymorphism, information about the structure of a program's state is not always explicit in the declarations. In the code fragment,

```
class Company {
   Person[] employees;
   ...
}
```

the relation between Company and Person can be trivially deduced. We can furthermore deduce that a Company may be related to zero or more Persons. However, suppose the employees relation were instead implemented as:

```
Vector employees;
```

Then the relation between Company and Person is not so easily derived. Many modern tools for reverse engineering an object model from code (such as Rational Rose [1]) would in this case show a relation between Company and Vector, labeled employees [1]. Some tools (e.g. Womble [9], the predecessor of Superwomble) use heuristics to correctly generate an object model for such an example. However, these techniques still cannot generate a reasonable model in this situation:

```
class Company {
   Hashtable departments;
}
```

Suppose departments contains a mapping from a department name (of type String) to a Vector of employees (which contains Persons) in that department. In this case, most tools would infer a relation between Company and Hashtable; Womble would infer relations between Company and String, and Company and Vector. The nesting of containers would not be correctly inferred. Such nesting is common in large programs, which are the kinds of program that would benefit most from automated analysis.

The essential problem in generating an object model in this case is to determine the runtime types of field references. But even if the type of object inside of a Vector or Hashtable could be determined, the resulting model would show the representation details of those classes. In a large system, this makes the model unwieldy; and this information is not always useful to the programmer, especially when a view of the design as a whole is required. In short, we would like the generated object models to be *abstract*. Additionally, we would like the tool to be *lightweight* enough that it can analyze large programs quickly, and *incremental*, so that it can be applied to partial programs.

The thesis describes a tool which addresses these problems. Specifically, this thesis will address the following questions:

- How can type analysis methods be applied to the problem of automatically generating code object models?
- How and to what extent can abstraction of an object model be automated?

Superwomble's technique for extracting object models is based on a type analysis that uses an unsound type system. This unsoundness is important because it allows for incrementality (a system can be analyzed even if some components of the system are not available for analysis). It is also useful for masking problems that arise due to other weaknesses of the analysis (namely, that it is flow insensitive). The essence of the analysis is this: for each method in a program, a type specification is generated; then for each call to that method, the effects of the method on the argument types and result type can be determined. Using this, the field types for a class can be determined; this is used to generate an object model. The extracted models are concrete; the tool includes facilities for automatically applying user-supplied abstraction rules to a concrete model.

This thesis is organized as follows. Chapter 2 gives an overview of the semantics and syntax of the object modelling notation that will be used throughout. Chapter 3 summarizes the intermediate language that will be used in subsequent chapters. Chapter 4 describes the type analysis in detail, while Chapter 5 describes how object models are generated from the resulting information. Chapter 6 discusses how the tool allows semi-automated abstraction of the object models. Chapter 7 discusses the uses of Superwomble-generated object models, and describes some case studies of Superwomble on various programs. Chapter 8 summarizes related work, both in terms of the type analysis and object model extraction tools.

Chapter 2

Object Model Syntax and Semantics

A brief introduction to object models is warranted, since that is the focus of this thesis. The object model syntax used is a subset of the Alloy modeling language [5]. The nodes in the object model represents sets of objects, which will correspond to Java classes and interfaces in the output produced by this tool. The labeled, directed edges represent associations in the object model. An edge from a node A to a node B labeled r represents a relation containing pairs of objects whose first object is in the set A and whose second object is in the set B. Unlabelled edges with triangular heads represent subtyping relationships (which correspond to subclassing and implementing interfaces). An example object model using this notation is given in Figure 2-3.

The ends of the arrows may contain additional multiplicity annotations. The possible markings are !, ? and *, which refer to exactly one, zero or one, and zero or more, respectively. The absence of a marking is equivalent to *, and indicates a multiplicity of zero or more. When there is a marking n at the B end of a relation r from A to B, this means that n B's are associated with each A under the relation r. For instance, the model in Figure 2 shows that each B is associated with m A's by r, and each A is associated with n B's by r [6].

For example, an edge from node A to node B marked ! at the A end means that each A is associated with exactly one B, but each B may be associated with many

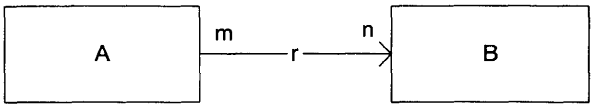

Figure 2-1: r associates each A with n Bs and each B with m As.

A's; that is, a function from A to B.

Finally, the ends of the arrows may contain mutability annotations, which show whether a relation is fixed over the life of the object. A small hatch through the end of an edge means that the relation is immutable with respect to the object at that end of the arrow. For instance, on an edge from A to B, a hatch through the B end means that the set of B's associated with a given A cannot change during the lifetime of that A:

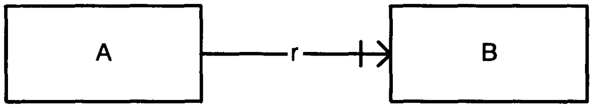

Figure 2-2: The set of Bs associated by r with an A is fixed over the lifetime of an A.

Superwomble infers multiplicities and mutabilities at the head end of associations, but not at the tails.

To summarize, we give an example of a simple object model that demonstrates these features. Consider a model of a class `Company`. There is a relationship *employees* from `Company` to a set of `Persons`. Each `Person` may work for any number of companies, which can change over time. The `Vector` used to store a company's employee list cannot change over time, but its contents can. A model of this system is shown in Figure 2-3.

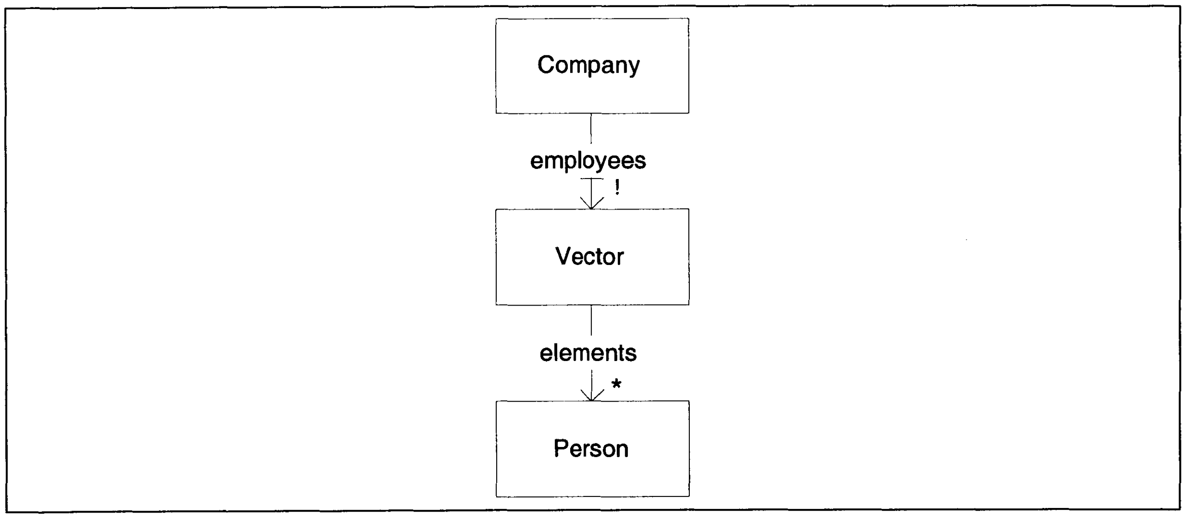

Figure 2-3: A sample object model, showing the relationship between a company and its employees.

Chapter 3

Language Definition

The implementation of this analysis works on Java bytecode. For the purpose of describing the analysis, we define a smaller language, which is a rough subset of Java bytecode. One notable feature that this language lacks is any instruction like `checkcast`, the instruction for type casts in Java bytecode. This is omitted because the Superwomble analysis does not use casting information.

3.1 Language Semantics

The program state is represented by a tuple $\langle pc, S, L, H, C \rangle$, where pc is the program counter, S is the stack, L store the local variables, H is the heap, and C is the call stack. L, C, and S are all represented as lists; H is a map. The heap maps object references to tuples containing the object's type and its fields. The fields are represented by a map from field names to object references. The operational semantics of the language are given in Figure 3-1. In the operational semantics, the notation $m[a : b]$ represents the state of m, with the addition of a binding from a to b (or if a is bound in m, the binding to b replaces that binding). The notation $::$ is used to denote consing to a list (e.g. $x :: L$ denotes the list containing containing x at the front followed by the elements of L).

Each method has only one return point; this restriction makes the type analysis in Chapter 4 easier to describe.

The operational semantics use the following helper functions:

- `init_object(c)` allocates space for a new object of class `c` onto the heap.
- `init_array_object(c,i)` allocates space for a new array whose elements have class `c` and capacity `i` onto the heap.
- `Dispatch(o, sig)` returns the location of the code for a method with signature `sig` in the runtime class of `o`.
- `calculate(op, v1, v2)` models the built-in operators; it calculates the value of applying the operation `op` to primitive values `v1` and `v2`. For instance, `calculate(+,1,2)` returns 3. This helper function is used in the operational semantics for `binary_op` to model the application of primitive operations.

In order to explain Figure 3-1, we describe the transitions for some sample instructions. The transition for `putfield` *f* shows that the top element of the stack (x) is the value that field *f* is to be set to, and that the next element on the stack (o) is the object whose field *f* is set. These two elements are popped from the stack and the value of o on the heap is changed so that field *f* has the value x.

The transition for `invoke_method` *sig* is a bit more complicated. The top n elements on the stack are the arguments to the method (in reverse order); the next element after those is the receiver of the method call. Dispatch(o, *sig*) is used to determine the location of the method that is to be called (the dispatch is dynamic – the method that is retrieved depends on the runtime type of o). The current program counter, stack, and local variable state are stored to the call stack. Then execution jumps to the location determined by Dispatch, with the stack initialized to an empty stack, and the local variable state initialized to contain the receiver and the arguments to the method.

At the return point of a method, there are two variants of the return instruction. `return` instruction pops the tuple from the top of the call stack. This tuple contains the program counter, stack, and local variable state that must be restored to continue execution at the program point where the method returns. `return_val` is used for

$$\frac{\text{instr(pc)} = \texttt{new\_object}\ c}{\langle pc, S, L, H, C\rangle \Rightarrow \langle pc+1, ref_{fresh} :: S, L, H[ref_{fresh} : init\_object(c)], C\rangle}$$

$$\frac{\text{instr(pc)} = \texttt{new\_array}\ c}{\langle pc, i :: S, L, H, C\rangle \Rightarrow \langle pc+1, ref_{fresh} :: S, L, H[ref_{fresh} : init\_array\_object(c, i)], C\rangle}$$

$$\frac{\text{instr(pc)} = \texttt{set\_local}\ k}{\langle pc, x :: S, L, H, C\rangle \Rightarrow \langle pc+1, S, L[k : x], H, C\rangle}$$

$$\frac{\text{instr(pc)} = \texttt{get\_local}\ k}{\langle pc, S, L[k : x], H, C\rangle \Rightarrow \langle pc+1, x :: S, L, H, C\rangle}$$

$$\frac{\begin{array}{c}\text{instr(pc)} = \texttt{putfield}\ f \\ H(o) = (classid, fields)\end{array}}{\begin{array}{c}\langle pc, x :: o :: S, L, H, C\rangle \\ \Rightarrow \langle pc+1, S, L, H[o : (classid, fields[f : x])], C\rangle\end{array}}$$

$$\frac{\text{instr(pc)} = \texttt{putstatic}\ f}{\langle pc, x :: S, L, H, C\rangle \Rightarrow \langle pc+1, S, L, H[f : x], C\rangle}$$

$$\frac{\text{instr(pc)} = \texttt{arraystore}}{\begin{array}{c}\langle pc, x :: i :: o :: S, L, H[o : (\text{Array}, fields)], C\rangle \Rightarrow \\ \langle pc+1, S, L, H[o : (\text{Array}, fields[elt_i : x])], C\rangle\end{array}}$$

$$\frac{\text{instr(pc)} = \texttt{getfield}\ f}{\begin{array}{c}\langle pc, o :: S, L, H[o : (classid, fields[f : x])], C\rangle \\ \Rightarrow \langle pc+1, x :: S, L, H[o : (classid, fields[f : x])], C\rangle\end{array}}$$

$$\frac{\text{instr(pc)} = \texttt{getstatic}\ f}{\langle pc, S, L, H[f : x], C\rangle \Rightarrow \langle pc+1, x :: S, L, H[f : x], C\rangle}$$

$$\frac{\begin{array}{c}\text{instr(pc)} = \texttt{arrayget} \\ \underline{H(o) = (\text{Array}, fields[elt_i : x])}\end{array}}{\begin{array}{c}\langle pc, o :: i :: S, L, H, C\rangle \\ \Rightarrow \langle pc+1, x :: S, L, H, C\rangle\end{array}}$$

Figure 3-1: Operational Semantics of Language

$$\frac{\mathrm{instr(pc)} = \texttt{invoke\_method}\ sig}{\begin{gathered}\langle pc, a_n :: \cdots :: a_1 :: o :: S, L, H, C\rangle \Rightarrow \\ \langle \mathrm{Dispatch(o, sig)}, \varepsilon, [0 : \mathrm{o}, 1 : \mathrm{a_1}, \ldots, \mathrm{n} : \mathrm{a_n}], \mathrm{H}, (\mathrm{pc} + 1, \mathrm{S}, \mathrm{L}) :: \mathrm{C}\rangle\end{gathered}}$$

$$\frac{\mathrm{instr(pc)} = \texttt{return}}{\langle pc, S, L, H, (pc', S', L') :: C\rangle \Rightarrow \langle \mathrm{pc'}, \mathrm{S'}, \mathrm{L'}, \mathrm{H}, \mathrm{C}\rangle}$$

$$\frac{\mathrm{instr(pc)} = \texttt{return\_val}}{\langle pc, x :: S, L, H, (pc', S', L') :: C\rangle \Rightarrow \langle \mathrm{pc'}, \mathrm{x} :: \mathrm{S'}, \mathrm{L'}, \mathrm{H}, \mathrm{C}\rangle}$$

$$\frac{\mathrm{instr(pc)} = \texttt{binary\_op}\ \mathrm{op}}{\begin{gathered}\langle pc, p_1 :: p_2 :: S, L, H, C\rangle \\ \Rightarrow \langle pc + 1, \mathrm{calculate}(op, p_1, p_2) :: S, L, H, C\rangle\end{gathered}}$$

$$\frac{\mathrm{instr(pc)} = \texttt{goto}\ x}{\langle pc, S, L, H, C\rangle \Rightarrow \langle x, S, L, H, C\rangle}$$

$$\frac{\mathrm{instr(pc)} = \texttt{cond\_jump}\ x}{\langle pc, \texttt{true} :: S, L, H, C\rangle \Rightarrow \langle x, S, L, H, C\rangle}$$

$$\frac{\mathrm{instr(pc)} = \texttt{cond\_jump}\ x}{\langle pc, \texttt{false} :: S, L, H, C\rangle \Rightarrow \langle pc + 1, S, L, H, C\rangle}$$

$$\frac{\mathrm{instr(pc)} = \texttt{dup}}{\langle pc, x :: S, L, H, C\rangle \Rightarrow \langle pc + 1, x :: x :: S, L, H, C\rangle}$$

$$\frac{\mathrm{instr(pc)} = \texttt{pop}}{\langle pc, x :: S, L, H, C\rangle \Rightarrow \langle pc + 1, S, L, H, C\rangle}$$

$$\frac{\mathrm{instr(pc)} = \texttt{push\_const k}}{\langle pc, S, L, H, C\rangle \Rightarrow \langle pc + 1, k :: S, L, H, C\rangle}$$

Figure 3-1: Operational Semantics of Language

methods that have a non-void return type; it restores the program state for the return point, and then pushes the methods return value onto the stack.

3.2 Examples

Throughout the description of the analysis, we will use part of an example `Vector` implementation to show how each step of the analysis works. The source code for the implementation is shown below. The `Vector` is represented as an array of `Objects`, and has methods for adding to the end of the vector, and setting an element in the vector.

```
public class Vector {
   private Object[] elements;
   private int size;

   public void addElement(Object elt) {
      if (size == elements.length) {
         Object[] newElts = new Object[elements.length*2];
         for (int i = 0; i<size; i++) {
            newElts[i] = elements[i];
         }
         elements = newElts;
      }
      elements[size] = elt;
      size++;
   }

   public void setElementAt(Object elt, int index) {
      elements[index] = elt;
   }

   public Object getFirst() {
      if (size == 0)
         return null;
      return elements[0];
   }
   ... }
```

Figure 3-2 shows this code after it has been transformed into the bytecode language described above. Notice that in the bytecode, the arguments are stored as the first local variables. The receiver of a method is regarded as argument 0 (which is also local 0). Each method has only one return point; in `getFirst`, there are logically two return points, but in the translated code, there is only one return point, and the different logical return points jump to this location. Also notice that `getFirst` uses `return_val`, which the other methods use `return`, since they have void return types.

```
addElement:
0:    get_local 0
      getfield size
      get_local 0
      getfield elements
      getfield length
      binary_op ==
      cond_jump 20
7:    get_local 0
      getfield elements
      dup
      getfield size
      get_local 1
      arraystore
      get_local 0
      dup
      getfield size
      push_const 1
      binary_op +
      putfield size
      return
20:   get_local 0
      getfield elements
      getfield length
      push_const 2
      binary_op *
      newarray Object
      store_local 2
      push_const 0
      store_local 3
28:   get_local 3
      get_local 0
      getfield size
      binary_op ≥
      cond_jump 45
33:   get_local 2
      get_local 3
      get_local 0
      getfield elements
      get_local 3
      arrayget
      arraystore
      get_local 3
      push_const 1
      binary_op +
      store_local 3
      goto 33
45:   get_local 0
      get_local 2
      putfield elements
      goto 7

setElementAt:
0:    get_local 0
      getfield elements
      get_local 2
      get_local 1
      arraystore
      return

getFirst:
0:    get_local 0
      getfield size
      push_const 0
      binary_op op
      cond_jump 10
      get_local 0
      getfield elements
      push_const 0
      arrayget
      goto 11
10:   push_const null
11:   return_val
```

Figure 3-2: Bytecode for `Vector` example

Chapter 4

Typing Rules

The core analysis of Superwomble is a type analysis of a Java bytecode program. In this chapter, we formally define the type analysis that is used and discuss some of the tradeoffs involved in the design of the analysis technique, weaknesses of the analysis, and notes on peculiarities of the implementation of the analysis in Superwomble.

Types are deduced for each program point in a method body. The type of a program point is an *environment type*, which gives types to the elements of the stack and the local variables at that program point. Figure 4-1 defines the language of type expressions. There are 3 core kinds of types in the language: variable types, method types, and environment types.

T represents variable types, which may represent object types or primitive types. An *object type* contains a Java class name, as well as a mapping from field names

$E \in \mathit{EnvironmentType} = (L, S)$
$L \in \mathit{LocalTypes} = \mathit{int} \rightarrow T$
$S \in \mathit{StackType} = T^*$
$T \in \mathit{VariableType} = \langle \text{class} : \mathit{string}, \text{fields} : \mathit{fieldTag} \rightarrow T \rangle \mid \texttt{primitive}$
$\mathit{fieldTag} = \mathit{string} \mid \texttt{*element*}$
$M \in \mathit{MethodType} = (T_{a_1}, T_{a_2}, \ldots, T_{a_k}, T_r)$
$m \in \text{method}$
$i, j, k, p \in \text{program point}$
$v \in \text{object}$

Figure 4-1: Language of Type Expressions

to the types of those fields. This mapping can also include the special field name `*element*` which is used to denote the content type of an array type. Primitive types are represented by the special value `primitive`, since the analysis does not try to deduce anything more specific about primitive types. L represents the local variable state at a program point, as a mapping from the local variable number to a variable type. S represents the stack, and consists of a list of variable types. A program point has one or more types consisting of the local variable state and stack state at that point – we call this an environment type. Finally, methods have types; a method with k arguments has a type that is a $(k+1)$-tuple consisting of the argument types and the return type.

For example, an object of class `C` with field `f` of class `F` (where `F` has no fields) would have the variable type

$$T = \langle \text{class} : \texttt{C}, \text{fields} : fTypes \rangle$$

$$fTypes = \{(\texttt{f}, T_f)\}$$
$$T_f = \langle \text{class} : \texttt{F}, \text{fields} : \{\} \rangle$$

A program point with two elements on the stack, whose types are T_{S_1} and T_{S_2} (where T_{S_2} is on the top of the stack, and with one local variable whose type is T_ℓ would have the environment type

$$E = ([T_\ell], [T_{S_1}, T_{S_2}])$$

The typing rules are given in Figure 4-2. There are three kinds of typing judgments: for program points (whose types are environment types), objects (whose types are variable types), and methods (whose types are *method types*).

The structure of the program point typing rules is as follows. Each instruction is subscripted by a pair of values that represent the entrance and exit program points for that instruction. For instance, $\textsf{dup}_{i,j}$ represents a $\textsf{dup}$ instruction at program point i that is followed by program point j. A type judgment is made for a program point j based on the type at program point i and the instruction. For a single instruction,

j is redundant, since it is always $i+1$. However, the second subscript is useful for composition of instructions. So a typing rule has the form

$$t_1 : T_1, \ldots, t_k : T_k, (\texttt{<instr>}\ args)_{i,j} \vdash j : U$$

and means that if the instruction at i is ($\langle$instr$\rangle$ $args$) and the type judgments $t_1 : T_1, \ldots, t_k : T_k$ all hold, then the type judgment $j : U$ holds. The typing rules for the instructions can only be applied for an instruction ($\langle$instr$\rangle$ $args)_{i,j}$ if that is the only instruction that is a predecessor to program point j. This is not explicit in the rule (for brevity). If a program point has more than one predecessor, then its type is derived from the types of all of its predecessor; this is described in the last rule.

The structure of object typing rules is straightforward: the typing judgment $v : T$ says that the object v has the variable type T. Similarly, a method typing rule $m : M$ says that the method m has method type M.

The typing rules contain a few helper functions:

- InitFields(c), which returns a function from field names to types, based on the declared field types in class c. For example, given the class definition

  ```
  class Person {
     String firstName, lastName;
     Date birthDate;
  }
  ```

  InitFields($Person$) would return the function

  $$\begin{array}{c}\{(\text{“}firstName\text{''}, \langle String, \{\}\rangle), \\ (\text{“}lastName\text{''}, \langle String, \{\}\rangle), \\ (\text{“}birthDate\text{''}, \langle Date, \{\}\rangle)\}\end{array}$$

- getMethod(sig) returns the method matching that signature (where a signature consists of the method name and declared type).

- returnPoint(m) returns the code position where the method returns (there is

$$i : (L, S), (\texttt{new\_object}\ c)_{i,j} \vdash j : (L, T :: S)$$
$$\text{where T} = \langle \text{c}, \text{InitFields(c)} \rangle$$

$$i : (L, \texttt{primitive} :: S), (\texttt{new\_array}\ c)_{i,j} \vdash j : (L, T :: S)$$
$$\text{where T} = \langle \text{Array}, \{(\texttt{*element*}, \langle \text{c}, \text{InitFields(c)} \langle\rangle\}\rangle$$

$$i : (L, T :: S), (\texttt{set\_local}\ k)_{i,j} \vdash j : (L[k : \text{merge}(L(k), T)], S)$$

$$i : (L, S), (\texttt{get\_local}\ k)_{i,j} \vdash j : (L, L(k) :: S)$$

$$i : (L, T_1 :: T_2 :: S), (\texttt{putfield}\ f)_{i,j} \vdash j : (L, S)$$
$$\text{where } T_1.\mathit{fields}(f) : T_2$$

$$i : (L, T_1 :: \texttt{primitive} :: \langle \text{Array}, \{(\texttt{*element*}, T_2), \ldots\}\rangle :: S), (\texttt{arraystore})_{i,j} \vdash$$
$$j : (L, S)[T_2 : \langle \text{Array}, \{(\texttt{*element*}, \text{merge}(T_1, T_2)), \ldots\}\rangle]$$

$$i : (L, T_1 :: S), (\texttt{getfield}\ f)_{i,j} \vdash j : (L, (T_1.\text{fields}(f)) :: S)$$

$$i : (L, \langle \mathit{Array}, \{(\texttt{*element*}, T), \ldots\}\rangle :: S), (\texttt{arrayget})_{i,j} \vdash j : (L, T :: S)$$

$$i : (L, \texttt{primitive} :: \texttt{primitive} :: S), (\texttt{binary\_op}\ \mathit{op})_{i,j} \vdash j : (L, \texttt{primitive} :: S)$$

$$i : (L, S), (\texttt{cond\_jump}\ k)_{i,j} \vdash j : (L, S)$$

$$i : (L, T :: S), (\texttt{dup})_{i,j} \vdash j : (L, T :: T :: S)$$

$$i : (L, T :: S), (\texttt{pop})_{i,j} \vdash j : (L, S)$$

$$i : (L, S), (\texttt{push\_const}\ k)_{i,j} \vdash j : (L, k :: S)$$

Figure 4-2: Typing Rules (Primitive Instructions)

$$\frac{\begin{array}{c}\text{returnPoint}(m) = r\\ 0 : ([T_{a_0}, T_{a_1}, \ldots, T_{a_k}], []), (\text{instrs}(m))_{0,r}\end{array}}{m : (T_{a_0}, T_{a_1}, \ldots, T_{a,k}, T_r)}$$

$$\frac{\begin{array}{c}i : E_0, (\mathit{instr}_1)_{i,j} \vdash j : E_1\\ j : E_1, (\mathit{instr}_2)_{j,k} \vdash k : E_2\end{array}}{i : E_0, (\mathit{instr}_1, \mathit{instr}_2)_{i,k} \vdash k : E_2}$$

$$\frac{\begin{array}{c}m = \text{getMethod}(\mathit{sig})\\ m : (T_{a_0}, T_{a_1}, \ldots, T_{a_n}, T_r)\end{array}}{i : (L, T_{a_n} :: \cdots :: T_{a_1} :: T_{a_0} :: S'), (\texttt{invoke\_method}\ \mathit{sig})_{i,j} \vdash j : (L, T_r :: S')}$$

$$\frac{\begin{array}{c}i : E_0, (\mathit{instr})_{i,j} \vdash j : E_1\\ \forall m_i \in \mathit{jumps}, m_i : E_{m_i}\\ \text{where } \mathit{jumps} = \{m \mid \mathit{instr}(m) = (\texttt{goto}\ k) \vee \mathit{instr}(m) = (\texttt{cond\_jump}\ k)\}\end{array}}{k : \text{merge}(E_1, E_{m_1}, \ldots, E_{m_n}), \text{where } n = |\mathit{jumps}|}$$

Figure 4-2: Typing Rules (Composite Rules)

only one return point, as described in Chapter 3).

- instrs(m) returns the sequence of instructions for method m.

4.1 Merging Types

A program point may have many types. For instance, at a join point in the control flow graph, there is a type deduced for each entry to that program point. The type of a program point that is eventually used as the output of the analysis is the most specific type that is compatible with all of the typing judgments made for that program point. This is the type that results from merging all of the types that have been found for that program point. The *merge* and *mergeEnv* helper functions are defined in Figure 4-3(in these definitions, we use $\ominus$ to denote symmetric difference).

The type that results from merging two types T_1 and T_2 depends on the relationship between T_1 and T_2 in the type hierarchy. If either type's class is a descendant in the declared subclass hierarchy of the other type's class, then the merged type's class

```
merge($T_1$, $T_2$) =

    if $T_1$.class is a subtype of $T_2$.class
        then $\langle$ $T_1$.class, mergeFields($T_1$, $T_2$) $\rangle$
    else if $T_2$.class is a subtype of $T_1$.class
        then $\langle$ $T_2$.class, mergeFields($T_1$, $T_2$) $\rangle$
    else $\langle$ mergeClass($T_1$.class, $T_2$.class), mergeFields($T_1$, $T_2$) $\rangle$

mergeClass($c_1$, $c_2$) =
    $c$ such that $c$ is the closest common ancestor of $c_1$ and $c_2$ in the Java type hierarchy

mergeFields($T_1$, $T_2$) =
    $T_1.fields \ominus T_2.fields \bigcup$
      $\{(f, merge(t_1, t_2)) \mid (f, t_1) \in T_1.fields \wedge (f, t_2) \in T_2.fields\}$

mergeEnv$(([\ell_1, \ldots, \ell_m], [t_1, \ldots, t_k]), ([\ell'_1, \ldots, \ell'_n], [t'_1, \ldots t'_k]))$ =
    $(L, S), where$
        if $m < n$
            $L = [merge(\ell_1, \ell'_1), \ldots, merge(\ell_m, \ell'_m), \ell'_{m+1}, \ldots, \ell'_n]$
        else
            $L = [merge(\ell_1, \ell'_1), \ldots, merge(\ell_n, \ell'_n), \ell_{n+1}, \ldots, \ell_m]$
        $S = [merge(t_1, t'_1), \ldots, merge(t_k, t'_k)]$
```

Figure 4-3: Definitions of merge operations

is the descendant type's class. If there is no such relationship between the classes of T_1 and T_2, then the merged type has the class of the closest common ancestor in the class hierarchy. So in one case, the class is found by walking up the type hierarchy, and in the other case, it is found by walking down the type hierarchy. The reason for these different cases is that when a type is merged with a subtype, we assume that is actually a polymorphic instantiation of the subtype. That is, we assume that the more specific type is what is really being used, but due to limitations of the type analysis, the more specific type has not be determined in one case. The intuition is that for any type that is instantiated to a more specific type than it is declared to, there are some parts of the program that know about the more specific types, and some that are generic; it makes sense to use the information in the parts of the program that knows the more specific type.

If the closest common ancestor were used in all cases, then if in any place in the code a field's type could not be reconstructed as a more specific type, then nothing would be gained from the parts of the analysis that did successfully construct a more specific type. This is problematic since some weaknesses of the analysis (for instance, flow insensitivity) result in overly general types to be deduced at some program points. But with this merging technique, as long as the more specific type is deduced at *some* program point, that information is available after the analysis is complete. If at some other place in the code, some other subclass is used, then that will subsequently be merged with the merged type, and the class will move back up the type hierarchy. For instance, if an object's type is determined to be `Object` in one place, and `String` in another place, then the types are merged to `String`. But if elsewhere in the code the type was determined to be `Integer`, then merging would climb back up the type hierarchy, and the merged type would be `Object`.

It is certainly possible that this strategy could produce incorrect results. The analysis might have correctly determined the most specific type of the supertype, because any instance of that supertype could appear in that field. In this case, the resulting types (and possibly the resulting object model) would be incorrect (the model would show some subset of the types that could really appear in a field). The

```
public class Student extends Person ...

public class University {
   private Vector employees;

   public void addEmployee(Person p) {
      if (p instanceof Student) {
         addStudentEmployee((Student)p);
      }
      else {
         employees.addElement(p);
      }
   }

   public void addStudentEmployee(Student s) {
      ...
      employees.addElement(s);
   }
```

Figure 4-4: A program fragment where unsound merging would affect Superwomble output.

type for a field would appear as one type when really it could be some other type (that shares a common supertype with the determined type).

The code in Figure 4-4 is an example of where the definition of merge would cause the output to not match the programmer's intentions. This code models a payroll system for a university; some employees may be students, and these employees may need to be handled specially (for instance, special accounting for federal work-study). Because of this special handling, there is one method that adds employees who are students. When the type contained in `employees` is determined to be `Person` at one program point, and `Student` at another program point, the merge operation resolves the contents to be of class `Student`. In fact, any type of `Person` may be contained in `employees`.

4.2 Examples

The method `setElementAt` in the `Vector` implementation described above provides a simple example of straight-line code that can be typed. Suppose that the arguments have the following types:

$$\mathit{this} : T_{\mathit{this}} = \langle \text{Vector}, \{(\mathit{size}, \texttt{primitive}), (\mathit{elements}, \langle \text{Array}, \{(\texttt{*element*}, T)\}\rangle)\}\rangle$$

$$\mathit{elt} : T_{\mathit{elt}} = T$$

$$\mathit{index} : T_{\mathit{index}} = \texttt{primitive}$$

Then the typing of the method can be achieved with the following steps:

$$\begin{gathered} 0 : ([T_{\mathit{this}}, T_{\mathit{elt}}, T_{\mathit{index}}], []), (\texttt{get\_local 0})_{0,1} \vdash \\ 1 : ([T_{\mathit{this}}, T_{\mathit{elt}}, T_{\mathit{index}}], [T_{\mathit{this}}]) \end{gathered}$$

$$\begin{gathered} 1 : ([T_{\mathit{this}}, T_{\mathit{elt}}, T_{\mathit{index}}], [T_{\mathit{this}}]), (\texttt{getfield } \mathit{elements})_{1,2} \vdash \\ 2 : ([T_{\mathit{this}}, T_{\mathit{elt}}, T_{\mathit{index}}], [\langle \mathit{Array}, \{(\texttt{*primitive*}, T)\}\rangle]) \end{gathered}$$

$$\begin{gathered} 2 : ([T_{\mathit{this}}, T_{\mathit{elt}}, T_{\mathit{index}}], [\langle \text{Array}, \{(\texttt{*primitive*}, T)\}\rangle]), (\texttt{get\_local 2})_{2,3} \vdash \\ 3 : ([T_{\mathit{this}}, T_{\mathit{elt}}, T_{\mathit{index}}], [\langle \text{Array}, \{(\texttt{*primitive*}, T)\}\rangle, \texttt{*primitive*}]) \end{gathered}$$

$$\begin{gathered} 3 : ([T_{\mathit{this}}, T_{\mathit{elt}}, T_{\mathit{index}}], [\langle \text{Array}, \{(\texttt{*primitive*}, T)\}\rangle, \texttt{*primitive*}]), (\texttt{get\_local 1})_{3,4} \vdash \\ 4 : ([T_{\mathit{this}}, T_{\mathit{elt}}, T_{\mathit{index}}], [\langle \text{Array}, \{(\texttt{*primitive*}, T)\}\rangle, \texttt{*primitive*}, \langle T, \{\}\rangle]) \end{gathered}$$

$$\begin{gathered} 4 : ([T_{\mathit{this}}, T_{\mathit{elt}}, T_{\mathit{index}}], [\langle \text{Array}, \{(\texttt{*primitive*}, T)\}\rangle, \texttt{*primitive*}, \langle T, \{\}\rangle]), \\ (\texttt{arraystore})_{4,5} \vdash 5 : ([T'_{\mathit{this}}, T_{\mathit{elt}}, T_{\mathit{index}}], []) \\ \text{where} T'_{\mathit{this}} = T_{\mathit{this}} \end{gathered}$$

This quite trivial example shows that if the receiver of the method contains objects of type T, and the element to be added to the vector is of type T, then right before returning from the method, the vector still contains objects of type T.

Suppose the receiver instead contained elements of type `String`, and that the method was called with an argument of type `Integer`. This case is more interesting, because in the typing rule for `arraystore`, the two types are merged. Since `String` is not an ancestor or a descendant of Integer in the type hierarchy, merging a String and an Integer with result in the type Object. So in this case, right before returning from the method, the receiver would contain Object in the field elements.

addElement provides a more interesting example of applying the typing rules, because of the control-flow constructs it uses. The control flow graph for this method is shown in Figure 4-5. The control flow graph shows where there are join points in the bytecode. Anytime there is new information about the type at the end of a basic block before a join point, the type for the basic block after the join point may have to be computed with this new information. We do this beacuse the rule for a program point with more than one predecessor says that we must compute the types at each predecessor to determine that program point's type. /*We do this to compute a fixed point on the type information. Computing the fixed point is essentially a heuristic for producing better results.*/ Let's say the method is invoked with a receiver whose type is:

$$T_{this} = \langle \text{Vector}, \{elements, \langle \text{Array}, \{(\texttt{*element*}, S\}\rangle\}\rangle$$

That is, the receiver's elements are objects of the class S. Suppose that the argument's type is T. Then at the start of the method, we have the typing:

$$0 : ([T_{this}, T_{elt}], [])$$

Using the typing rules for straight-line code allows us to find the type at the end of the first basic block:

$$0 : ([T_{this}, T_{elt}], []), (instr[0..6])_{0,7} \vdash 7 : ([T_{this}, T_{elt}], []), 20 : ([T_{this}, T_{elt}], [])$$

From the type at program point 20, we apply rules to determine the type at the end of that basic block:

$$\begin{gathered} 20 : ([T_{this}, T_{elt}], []), (instr[0..6, 20..27])_{0,28} \vdash \\ 28 : ([T_{this}, T_{elt}, \langle \text{Array}, \{(\texttt{*element*}, \langle \text{Object}, \{\}\rangle)\}\rangle, \texttt{*primitive*}], []) \end{gathered}$$

Likewise,

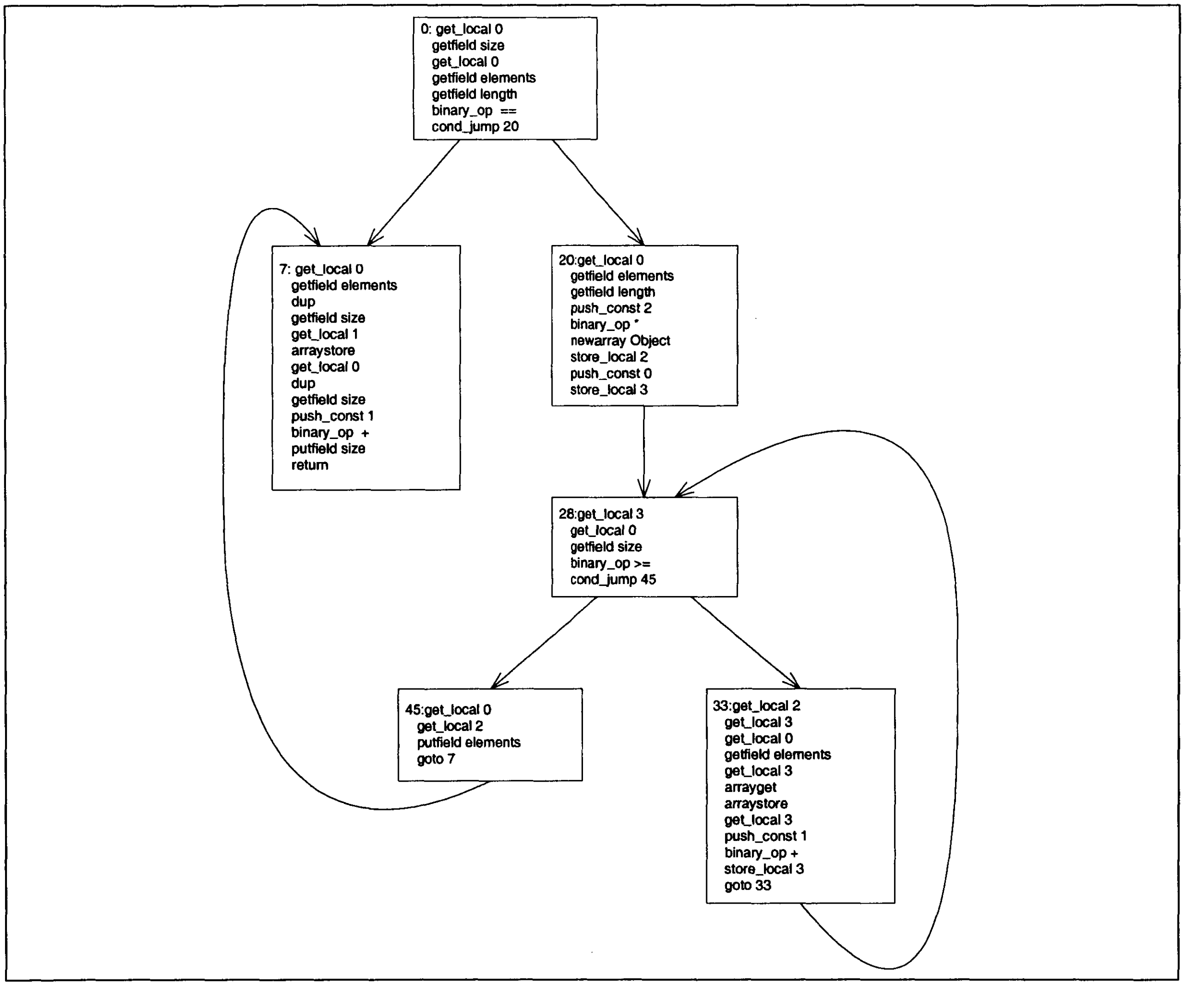

Figure 4-5: Control flow graph for `Vector.addElement`.

$$\begin{gathered}28 : ([T_{this}, T_{elt}, \langle \text{Array}, \{(\texttt{*element*}, \langle \text{Object}, \{\}\rangle)\}\rangle, \texttt{*primitive*}], []),\\ (instr[0..6, 20..32])_{0,33} \vdash\\ 33 : ([T_{this}, T_{elt}, \langle \text{Array}, \{(\texttt{*element*}, \langle \text{Object}, \{\}\rangle)\}\rangle, \texttt{*primitive*}], []),\\ 45 : ([T_{this}, T_{elt}, \langle \text{Array}, \{(\texttt{*element*}, \langle \text{Object}, \{\}\rangle)\}\rangle, \texttt{*primitive*}], [])\end{gathered}$$

$$\begin{gathered}33 : ([T_{this}, T_{elt}, \langle \text{Array}, \{(\texttt{*element*}, \langle \text{Object}, \{\}\rangle)\}\rangle, \texttt{*primitive*}], []),\\ (instr[0..6, 20..44])_{0,45} \vdash\\ 28 : ([T_{this}, T_{elt}, \langle \text{Array}, \{(\texttt{*element*}, S)\}\rangle, \texttt{*primitive*}], [])\end{gathered}$$

At this point, there are two types for program point 28. Since we want to find a fixed point for the types, we handle this loop by reconsidering the types from program point 28 on. We merge these types to find that

$$28 : ([T_{this}, T_{elt}, \langle \text{Array}, \{(\texttt{*element*}, S)\}\rangle, \texttt{*primitive*}], [])$$

Based on this type, we find new types for program points 45 and 33. The new type for 33 is the same as the one that already existed, so we do not need to merge the types or retype the basic block starting at 33. The new type for 45 does differ, and is merged with the old type to:

$$45 : ([T_{this}, T_{elt}, \langle \text{Array}, \{(\texttt{*element*}, S)\}\rangle, \texttt{*primitive*}], [])$$

Typing the basic block starting at 45 gives us a new type at 7:

$$7 : ([T_{this}, T_{elt}, \langle \text{Array}, \{(\texttt{*element*}, S)\}\rangle, \texttt{*primitive*}], [])$$

We merge this with the other type at 7 to get:

$$7 : ([T_{this}, T_{elt}, \langle \text{Array}, \{(\texttt{*element*}, S)\}\rangle, \texttt{*primitive*}], [])$$

Finally we type the basic block starting at 7 to find the type right before returning:

$$\begin{gathered}return : ([\langle \text{Vector}, \{elements, \langle \text{Array}, \{(\texttt{*element*}), V\}\rangle\}\rangle, T_{elt},\\ \langle \text{Array}, \{(\texttt{*element*}, V)\}\rangle, \texttt{*primitive*}], [])\\ \text{where U} = \text{merge}(S, T)\end{gathered}$$

So the final type of the Vector's elements is the type that arises from merging S and T. So, if we add a String to a Vector of Objects (or more precisely, a vector where no more specific type could be derived by the analysis), the resulting type is a Vector whose elements are Strings. If, on the other hand, we add String to a Vector of Integers, the resulting type is a Vector whose elements are Objects.

4.3 Weaknesses of the Analysis

This analysis is flow insensitive. The type of a variable consists of the merged type for all possible types that that variable could have during a method's execution. As a result, the analysis may give less specific type information than could be inferred from reading the program text. For example, consider the code sequence:

```
Object o;
Vector vec = new Vector();
if (a instanceof String) {
   o = (String)a;
   vec.addElement(o);
   ...
}
else {
   o = (Integer)a;
   ...
}
```

From reading this piece of code, we can infer that vec contains elements of type String. However, since the analysis is flow insensitive, the type of o at every point in the method is an Object (the result of merging a String and an Integer). So the type in vec is inferred to be Object, not String.

This behavior has been observed in the analysis of some programs (in fact the example code was inspired by code from a program where this was observed). That is, through instrumenting the implementation, it is possible to find that in some places, the flow insensitivity results in less precise types. However, even in these cases, the resulting object model has not been affected. This is a benefit of the merging technique used. In some parts of the program, due to limitations of the analysis, the type that is reconstructed for an object may be less specific than the type reconstructed at another point in the program. In these cases, the merge operation will resolve the type to be the more specific type. For instance, in the example above, the type in vec would be determined to be Object at this program point; but in some other program point it could be determined to be String. The merging operation would resolve vec to contains Strings.

4.4 Implementation Notes

The implementation of this analysis in Superwomble is a fairly straightforward translation of these rules. It works as follows. For each method, the code is analyzed using type variables for its arguments. From this analysis, a template for the method's type is created. The template describes how the method call effects the types of the arguments and what the result type is, based on the argument types. For instance, for the method

```
public void setBinA(A a, B b) {
   a.b = b;
}
```

the template would include information that the type of the field b in argument 1 has the type of argument 2. Then for each call site, the template is instantiated with the argument types for that call. So in the example above, if b has type T_b, then field b in the first argument must have type T_b also. In all other ways, the implementation is a direct translation of the rules presented above.

We now discuss a few interesting implementation choices in Superwomble.

4.4.1 Method Dispatch

In the typing rules, getMethod is used to determine how a method call affects the types in the method being analyzed. The method whose type is used is the method that is declared in the code. That is, the dispatch on the typing of a method is static. During the course of the analysis, more information about the type of the receiver of a method call may be available. If the receiver is known to have a more specific type than what is declared (that is, a subtype of what is declared), then it would be a more accurate approximation to the runtime type behavior if the type analysis were dispatched on the more specific type. However, this type of dispatching is not used due to limitations in the implementation.

Recall that the implementation uses template types so that the bulk of the analysis in a method is done once and cached. So at the time that the template is created (which is when method calls in that method are dispatched), the "dynamic" types of local variables are unknown. It is possible that some more specific type information could be obtained during this phase of the analysis (that is, more specific that the types declared by the programmer, but still possibly less specific than the runtime types). It is unclear if the availability of this information would arise that often. And even if it did, it may yield inconsistent results to choose a type to dispatch on that is neither the static type nor the runtime type, but possibly some intermediate type.

An alternative method for implementing the type rules is to inline all of the calls in the program; that is, for each method invocation, the code of that method could be analyzed from scratch, using the argument types derived at the call site. This would be quite inefficient, since typing a method requires typing each method that it calls, and the methods that they call, etc. It is much more efficient to analyze a method once and cache the results, only changing what changes across invocations of the method. It might be possible to cache some of the work required to type a method, without forcing the method invocations to be resolved until the types on that invocation are know. However, empirically it seems that most of the work in typing a method is spent while typing the methods that it calls, so little of the computation could be done without knowing what methods to dispatch on. Dispatching based on static types is certainly the simplest technique. And in practice, no cases have been observed where this led to defects in the final output.

One problem with using the methods based on the static types is what to do when the static type is an interface or an abstract method. In Superwomble, this will lead to not finding the method upon lookup, and thus the return value is as declared, and there are no additional type constraints on the arguments (beyond the declared types of the arguments).

4.4.2 Approximations in the Control Flow Graph

When control-flow graphs are generated from the bytecode, a few approximations are made, which could affect the results of the analysis. Java bytecode includes an instruction `jsr` for subroutine calls. A subroutine is jumped to, and then the subroutine returns to a location based on the value of a local variable. On the subroutine call, the return value is set to the next instruction after the subroutine call instruction. In the subroutine call, the local variable could be changed to any value. In Superwomble, the return address from the subroutine is assumed to not change in this way.

Exception handling is also not handled in the generation of control-flow graphs. A method call is always assumed to terminate normally, so exception-handling code is never considered in the type analysis. This approximation is used for the following reason. It would be incorrect to model an exception as a regular exit point from a method, because when a method exits with an exception, it returns to a different location in the calling code than a normal exit. In order to avoid these complications, Superwomble does not handle the control constraints imposed by exception handling.

4.4.3 Recursion

The type rules for method calls require determining the type of the method being called, based on arguments on the stack. This presents a problem for recursive calls. In Superwomble, recursive calls are not dispatched on. That is, there are no type constraints on the arguments for that method call (beyond the declared types). The return type that is pushed onto the stack has the type that is declared to be the return type of the method. A better approximation may be to unroll the recursion to some fixed depth. This is a possible extension to Superwomble. There is no evidence that the current handling of recursion causes problems in the object models that are generated.

4.4.4 Approximating the Type Hierarchy

The Java type hierarchy of the classes in the program is necessary in order to merge types. Superwomble does not require that all classes in a program be available in order to run. As a result, some of the classes in the type hierarchy might not be available, so a complete type hierarchy cannot be determined. For instance, a class `A` might be available for analysis, and it might subclass a class `B`. If the code for `B` is not available, there is no way to know if `B` is a subclass of some other class. In these cases, we assume that `B`'s superclass is `Object`. This approximation has caused observable defects in the object models generated for some partial programs.

4.5 Extensions

We discuss some possible extensions to the current analysis framework.

4.5.1 Union Types

Two types T_1 and T_2 are merged when a variable is deduced to have either type T_1 or type T_2. Some information is clearly lost in merge operations that walk up the type hierarchy. After the types are merged, the variable has one type. So merging two types leads to a type that may be more general than what the type could ever really be (in the case when the merged type is a supertype of T_1 and T_2). An alternative way to handle merging types is to define union types, so that a type can express that its type is either T_1 or T_2. The technique would make the analysis much more complicated overall. In terms of implementation, this would increase space usage, since a union type would have to carry around all of the types that it is made up of, as well as all of their field types. The running time of the analysis would also probably be affected, since a tree would have to be traversed to find the type of a field in a union type.

If union types were used during the analysis, they would eventually need to be converted to a type to be displayed in an object model. It would be rather unwieldy to graphically display nodes in the object model as being of one type or another. The same type of merging operation used here could be applied to postprocess the types derived from the analysis. Merging into a supertype versus storing union types (and delaying the merge operation) could affect the typing of variables in subsequent parts of the analysis.

4.5.2 Dynamic Dispatch

The choice to use static dispatching is related to the choice to not use union types to merge types. Suppose we did use dynamic dispatching in the analysis. Then if two types A and B have been merged to type C, any methods called on that object would be dispatched on C. So C's implementation would be applied. If we stored union types, though, then we could dispatch A's method on A, and B's method on B. This could lead to more accuracy.

For example, say that we stored a union type of A and B, where

$$A = \langle \text{class} : C_A, \text{fields} : (f1, T_1) \rangle$$

$$B = \langle \text{class} : C_B, \text{fields} : (f2, T_2) \rangle$$

Suppose that a method m is called on this union type. C_A's implementation m sets field $f2$ to the value in $f1$; C_B's implementation of m sets field $f1$ to the value in $f2$. Then if we apply the two methods to the two types in the union, we get

$$A = \langle \text{class} : C_A, \text{fields} : (f1, T_1), (f2, T_1) \rangle$$

$$B = \langle \text{class} : C_B, \text{fields} : (f2, T_2), (f1, T_2) \rangle$$

If we had merged A and B to some common supertype instead, this level of detail could not be inferred from the analysis.

We could use dynamic dispatch of methods if a method were re-analyzed for each call site. However, this would likely cause a large slow-down in the analysis. And without union types, this technique is not likely to produce much better results. As discussed earlier, union types would increase space usage; so in order to improve the method dispatch mechanism, both space and time utilization would grow. Since one of the goals of Superwomble is to be lightweight, using the less effective techniques is a good tradeoff.

Chapter 5

Generating Object Models

In the last chapter, we described the type analysis that is applied to Java bytecode programs as the first step in Superwomble's analysis. From the type information that is gathered from that analysis, we generate an object model whose nodes are Java classes and interfaces and whose edges correspond to the fields of the Java classes.

The type analysis described in the Chapter 4 constructs an environment type for each method, given an input environment type. Using this, we can construct an object model of class C as follows.

- For each method m in C (with method signature $m(a_1, a_2, \ldots, a_n)$), define $T_{a_k} = \langle \text{class} : a_k, \text{fields} : \{\}\rangle$, and define T_m such that $m : (T_m, T_{a_1}, \ldots, T_{a_n})$.
- Define $T = \langle \text{class} : \texttt{C}, \text{InitFields}(C)\rangle$.
- For each method m, $T \leftarrow \text{merge}(T, T_m)$.
- Construct object model of T using its field types.

For example, we can derive the field types in `Vector` as follows. First, we determine the type of each method based on the default input environment; that is, the environment type consisting of the declared argument types (and the declared field types for the receiver). For `addElement`, the default environment is:

$$E = ([\langle \text{Vector}, \{(\textit{elements}, \langle \text{Array}, \{(\texttt{*element*}, \langle \text{Object}, \{\}\rangle)\}\rangle,)\}\rangle, \langle \text{Object}, \{\}\rangle], [])$$

From the typing of `addElement` in Chapter 4, we know that if at the start of the method, `elements` contains T's, and the argument is of type S, that at the end, `elements` contains $merge(S,T)$'s. So given the starting environment E, at the end of the method, `elements` has the same type as at the start.

For `setElementAt`, the default environment is:

$$E = ([\langle \text{Vector}, \{(\textit{elements}, \langle \text{Array}, \{(\texttt{*element*}, \langle \text{Object}\{\}\rangle)\}\rangle,)\}\rangle, \\ \langle \text{Object}, \{\}\rangle, \texttt{*primitive*}], [])$$

From the typing of `setElementAt` in Chapter 3, we know that this starting environment will result in `elements` containing `Objects` at the end of the method. If we merge this result with the result from typing `elementAt`, we find that `elements` contains `Objects`. That is, the type analysis has not provided any more information than the field declaration alone. A model of this is shown in Figure 5-1(a).

The resulting description of field types is from that class's point of view. That is, the type of a field in the class may actually depend on the use of that class (in `Vector` for example). Each class has its own view of the object model, and an object model of an entire system would consist of the view of some top-level class or classes. The object model from one class's perspective is extracted from the types of its fields. Consider the class `Company`, which uses `Vector`.

```
class Company {
   Vector employees = new Vector();

   public void addEmployee(Person p) {
      employees.addElement(p);
   }
}
```

In this case, the type of `employees` will contain `elements` of type `Person`. That is, employees has type T, where:

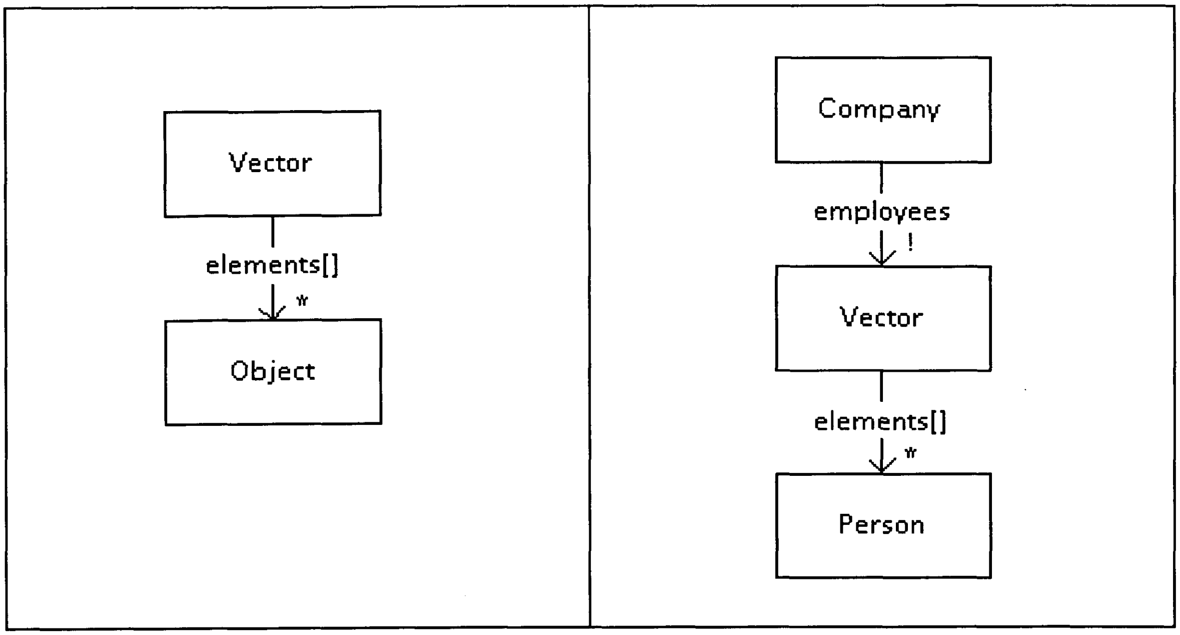

Figure 5-1: (a) Object model of `Vector`. (b) Object model of `Company`, which uses `Vector`.

$$T = \langle \text{Vector}, \{(elements, T2)\}\rangle$$

$$T_2 = \langle \text{Array}, \{(\texttt{*element*}, T_3)\}\rangle$$

$$T_3 = \langle Person, \{\}\rangle$$

This description can be used to create the object model in Figure 5-1. The object model is generated from the type by first creating a node in the model for the top-level type's class (`Vector`). Then, each of the types of its fields are visited and a node is created for those types. Superwomble automatically elides arrays, to show an edge to the content type of an array (with a '*' multiplicity on the head). So no node is created for T_2 with class `Array`; instead, a node is created for T_3 labelled `Person`. An edge is added from the first node (`Vector`) to the other node (`Person`), and it is labelled with the name of the field `elements`. Then the new nodes that have been added (in this case `Person`) are expanded in a similar way. So the fields of each type are followed until the model is complete.

Using this algorithm to generate object models results in each use of a class appearing as its own node, and the entire subgraph below it (corresponding to its fields) appears for each use. This leads to prohibitively large graphs. After this initial graph has been generated, Superwomble collapses identical nodes (and their subgraphs) into one; we call this *node folding*. If two nodes represent the same class and their fields are all equivalent (that is, they represent the same class, and their fields are equivalent), then those two nodes can be folded into one (and so can their children).

5.1 Object Model Annotations

So far we have only discussed the analysis that is necessary to determine the types of a class's fields; that is, the nodes and edges in the object model. Superwomble also includes multiplicity and mutability annotations in the object models that it generates. These annotations are determined based on a straightforward dataflow analysis during the type analysis. Mutability is determined based on whether a field is set in the code that is analyzed. If a field is set on any code path that is not in a constructor for the class, then that field is deemed to be mutable. Determining whether a field is mutable is context-specific. In one use of a class, a field may be mutated, while in another use, it may never be changed; in the former case, the field would be determined to be mutable, while in the latter case, it would be deemed immutable.

Multiplicities are determined as follows. If a field is an array, then that field has a multiplicity of zero or more by default. For all other fields, the multiplicity can be either zero or one ('?') or exactly one ('!'). A multiplicity is set to zero or one if that field is ever set to null. This is determined during the type analysis, by detecting what fields may be set to null. Like the type analysis, this is context-specific – the multiplicity of a field in some class C may depend on the top-level class being analyzed (that is, it depends on how class C is used in that specific context). If no code path sets a field to null, then the multiplicity is set to exactly one. A multiplicity of '*' is obtained through abstraction rules (described in Chapter 6) and the automatic abstraction of arrays.

Superwomble makes no attempts to infer multiplicities and mutabilities on the tail ends of relation arrows. To accurately detect either of these annotations would require an alias analysis.

In addition to the default multiplicities and mutabilities described above, in the user-defined abstraction rules described in Chapter 6, the user can specify the mutability or multiplicity for an abstract relation.

5.2 Tool Features

Superwomble offers some additional features for manipulating the object models that it generates. The object models are visualized in a graphical format that deviates slightly from the notation in this thesis (see Chapter 7 for a description of the differences), due to a limitation of the graph drawing tool. In order to generate an object model, the user first chooses which classes to include as the top-level classes in the model. Then the user has a choice of several options for what is included in the model – object model relations, type hierarchy edges, or both maybe included; and the user can choose to restrict the model to only those classes selected or to exclude Java core library classes from the model. Once the model has been generated, the use can select nodes in the model and redraw the subgraph of the model restricted to those nodes.

5.3 Extensions

Given the type analysis framework described in Chapter 4, there are not really any alternatives to how to generate the object model. There are, however, many possible extensions to the tool itself. One possible extension to Superwomble would be to allow a more flexible way for users to specify the model to be generated; for instance, it might be useful if a user could specify the top-level classes to include, as well as a maximal set of classes to be included in the model. Another useful option would be to exclude inheritance edges corresponding to implementing interfaces (that is, to include only subclassing edges), or to exclude inheritance edges arising from inheritance of `java.lang.Object`.

Chapter 6

Abstraction

The object models that are generated as described in Chapter 5 often reveal more implementation detail than the user is interested in. A more abstract model, especially for code using library classes, is often useful when trying to gain an overview of a system's structure.

Superwomble allows the user to define abstraction functions to be applied to an object model. An abstraction function is essentially just a rewrite rule that allows users to hide implementation details. In addition to the user-defined abstraction functions, some rules for commonly used core API classes are provided.

6.1 Examples

The Java library class `Hashtable` provides an example of a commonly used class for which an abstraction function is useful. The Java library class `Hashtable` is represented as follows:

```
class Hashtable {
   HashtableEntry[] table;
}

class HashtableEntry {
   Object key;
   Object value;
}
```

From an abstract perspective, the Hashtable should look like a set of key-value pairs. The following abstraction function maps the concrete representation of a Hashtable to this abstract view:

$AF(c) = a|\ a.pairs = \{(a.key, a.value)|a \in ht.table.elements\}$

The abstraction function is read as follows. Consider c to be the concrete value of a Hashtable; the abstraction function takes such a concrete value as an argument and returns a, the abstract value corresponding to it. The abstract value consists of *pairs*, a set of key-value pairs, which are defined to be the pairs is *ht.table.elements*.

This function can be used as a tree-rewriting rule in order to transform the default object model into a more abstract object model. For example, the program:

```
class Company {
   Hashtable salaries = new Hashtable();

   void addEmployee(Person p, Salary s) {
      salaries.put(p, s);
   }
}
```

would by default generate the object model in Figure 6-1(a). But we might prefer an object model with fewer implementation details in it, for two reasons. First of all, the implementation details of Hashtable may clutter the model. Second, the user might want to know about the abstract state of the program, rather than a particular implementation – in this case, salaries could be represented in many different ways, such as a Hashtable, a Vector of Person-Integer pairs, or two parallel linked lists. But

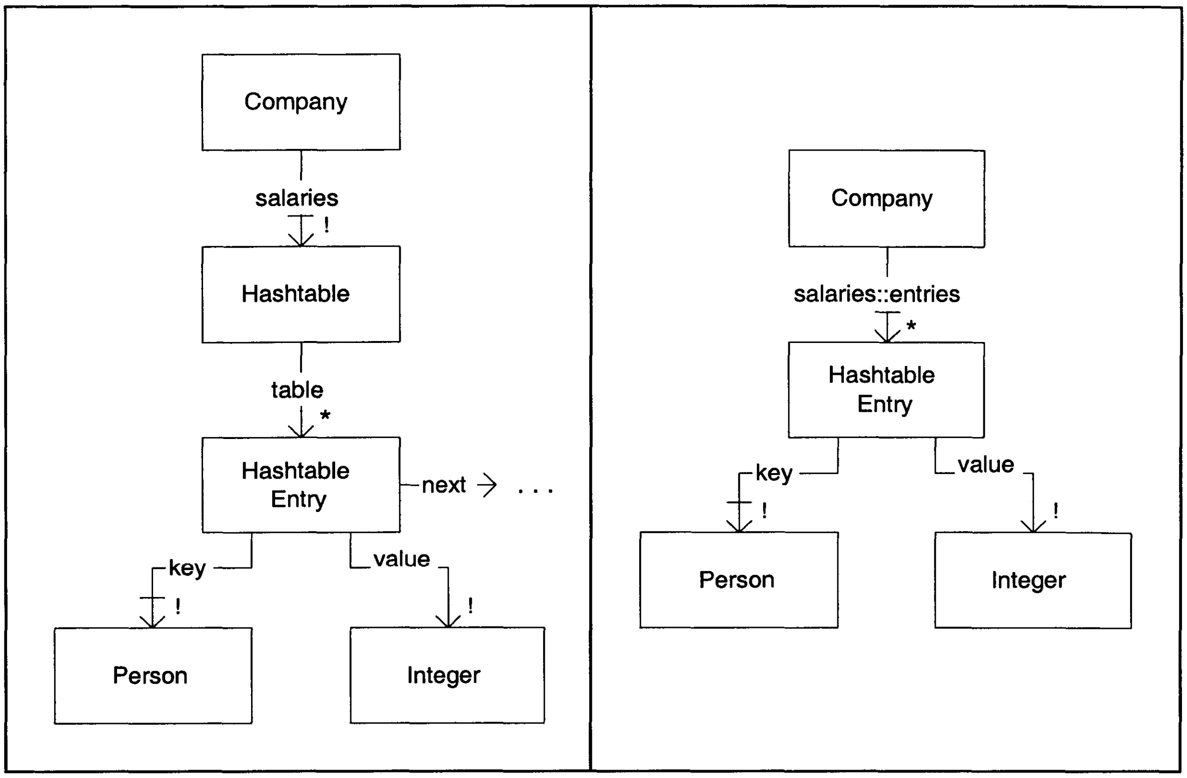

Figure 6-1: Object model of `Company` (a) without abstraction (b) with abstraction of `Hashtable`.

the user may only care that it is a set of pairs. Using the rewriting rule above, the object model in Figure 6-1(b) could be generated.

We now describe Superwomble's language for abstraction functions, by providing examples of using this language, followed by a formal description of its syntax and interpretation.

Consider the alternate implementation of `Company` that just contains a `Vector` of employees.

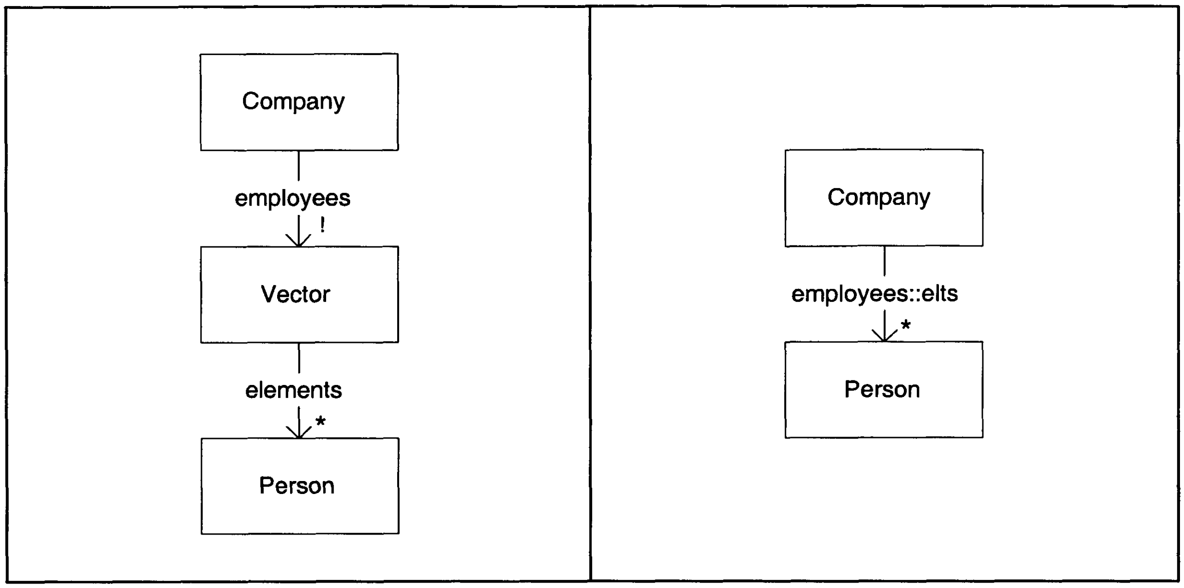

Figure 6-2: Object model of `Company` (a) without abstraction (b) with abstraction of `Vector`.

```
class Company {
   Vector employees;

   void addEmployee(Person p) {
      employees.addElement(p);
   }
}
```

Assume that the `Vector` implementation has the same data representation as the implementation introduced in Chapter 3. Figure 6-2 shows two different object models for the code – the model that Superwomble would construct by default, as well as a model where `Vector` has been abstracted away.

6.2 Example Abstraction Functions

Before discussing the formal syntax and interpretation of the abstraction rules, a few examples will be discussed.

The example rewrite given in Figure 6-3 is achieved by using the rule:

```
o: java.lang.Object | o.(java.util.Vector v).elementData -> o.v::elts
```

This rule elides `Vectors` and replaces them with a relation from the class which contains the `Vector` to the class contained in the `Vector`. The rule says that for any object `o` that is an instance of `java.lang.Object` (i.e. any object) and has a field of type `java.util.Vector`, the path consisting of the edge to the `Vector` and the edge from the `Vector` called `elementData` should be replaced by one edge from `o`, whose name is derived by concatenating the name of the field referred to by `v` with "`::elts`". In Figure 6-3, this rule hides the implementation choice of a `Vector` for `employees` and shows the more abstract representation of `employees` as a set of `Persons`.

There are two rewrite rules for `java.util.Hashtable` as in Figure 6-1:

```
o: java.lang.Object | o.(java.util.Hashtable h).table -> o.h::entries
h: java.util.HashtableEntry | h.next -> NULL
```

The first rule elides the `Hashtable`, and the second rule removes the edge `next` from each `HashtableEntry` (`NULL` is a special value to be used to remove a sequence of edges). After these two rules are applied, a relation that is implemented as a `Hashtable` instead looks like the more abstract set of key-value pairs.

6.3 Syntax and Interpretation of Abstraction Rules

The grammar for abstraction rules is shown in Figure 6-3.

The `vars` at the beginning of the rule, and on each side of the arrow in each rewrite must be the same. The rule says that for each object `var` of class `type`, the sequence of `rewrites` is to be applied. A `rewrite` says that a sequence of edges described by the sequence of `field_descs` (ending at object `dest`) should be rewritten to a single edge from `var` to `dest`, which is named by concatenating the names of some of the field variables that were resolved in the original sequence, with a name label.

```
rule ::= var ':' type '|' rewrite ('&&' rewrite)*
rewrite ::=  var ('.' field_desc)* '->' var ('.' field_var)* '::' label mult |
             var ('.' field_desc)* '->' NULL
field_desc ::=  field_name |
                '(' type field_var mult ')'
mult ::=  '?' |
          '!' |
          '*'
type ::= package-qualified Java class name

label, field_var, var ::= string
```

Figure 6-3: Syntax of Abstraction Language

Alternatively, if the special symbol `NULL` is on the right-hand side of a `rewrite`, then the sequence of edges on the left-hand side are removed (and replaced with nothing).

If a multiplicity is specified in a `field_desc` on the left-hand side, then only an edge with that multiplicity will be matched. When a multiplicity is specified on the right-hand side, any existing multiplicity for that edge is overridden with that value. By default, the multiplicity and mutability on an edge derived from a rule is calculated based on the multiplicity and mutability values of the replaced edges (for example, if one edge in the sequence has the multiplicity *, then the resulting edge will have that multiplicity; or, if some edges in a sequence have multiplicity ! and all others have multiplicity ?, the resulting multiplicity would be ?).

A generic example of the abstraction language is the set of rules:

```
a: A | a.(B b)?.c -> a.b::c
a: A | a.(C c)! -> NULL
```

Figure 6-4 shows how these rewrite rules would affect a sample object model. Since a multiplicity is specified in the first rule, the *b*2 and *c* edges are collapsed into the edge *b*2 :: *c*, but *b*1 is not changed. Furthermore, the resulting edge from `A` to `C` (*b*2 :: *c*) is not elided, because it does not have the '!' marking required by the second rule.

When several rewrite rules are separated by '&&', the edge sequence that is to be

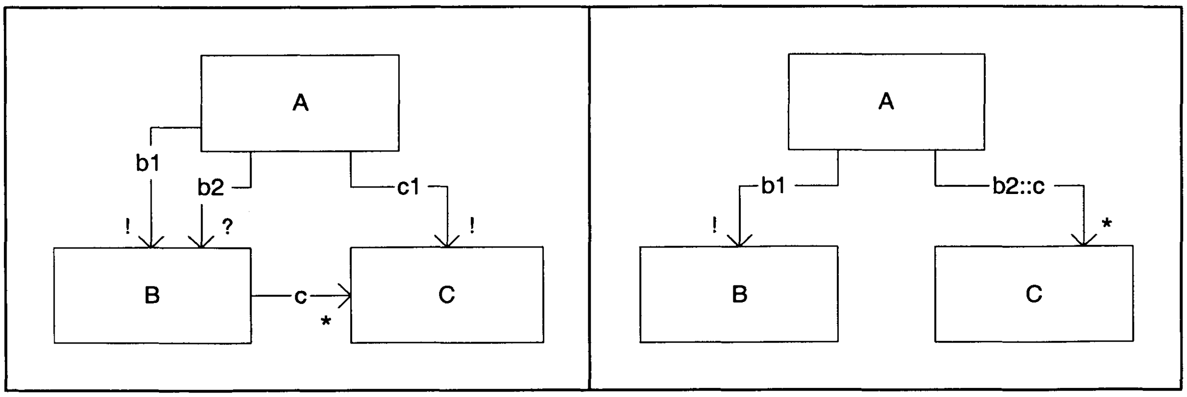

Figure 6-4: Generic example of an object model (a) before abstraction, and (b) after abstraction.

rewritten in each rule is not removed until all of the rewrite rules in the rule have been applied. This allows multiple rules that replace the same sub-pattern to be applied simultaneously.

For example, the model in Figure 6-1 could be further abstracted to elide HashtableEntry, so that Hashtables are represented as a set of keys and a set of values. The following rules could be used:

```
o: java.lang.Object | o.(java.util.HashtableEntry h*).key -> o.h::keys &&
                      o.(java.util.HashtableEntry h*).value -> o.h::values
```

This rule will result in edges from Company to Person and Company to Integer, which correspond to keys and values. (Add a figure to show this.) Note that the following two rules would not work correctly:

```
o: java.lang.Object | o.(java.util.HashtableEntry h*).key -> o.h::keys
o: java.lang.Object | o.(java.util.HashtableEntry h*).value -> o.h::values
```

In this case, the first rule would be applied, and the edge from Company to HashtableEntry would be removed, so that the second rule would not match anything.

6.4 Extensions

The abstraction language supported by Superwomble allows only a very limited type of abstraction. A sequence of edges can be collapsed into a single edge, which is especially useful for filtering out representation details that the user is not interested in (and which clutter the model). However, in general, an abstraction function can include much more complex types of abstraction. The rules allowed by Superwomble do not allow a user to automatically generate an abstract object model by applying a general abstraction function. In this section we describe some additional types of abstraction that would be useful for Superwomble to handle. We do this by describing example code where different abstraction techniques are desirable.

Consider an implementation of a graph data type. There are many different ways to represent a graph, and there are also different ways to abstractly describe the state of a graph. Two different representations are described in Figure 6-5. Suppose we stored `Strings` as the nodes of a graph. For these two representations, Superwomble would generated the object models in Figure 6-6 (without any sort of abstraction applied).

The abstract state of a graph can be described in many ways. Sometimes different descriptions of abstract state are just a matter of taste; in other cases, they actually express different things. Figure 6-7 shows three different abstract object models for a graph (these abstract models were inspired by course materials for 6.170 [3]). In the first model, a graph is regarded as a set of nodes and an adjacency relation *adj* between nodes. In the second model, a graph consists of a set of links (representing edges) and nodes; each link has a *from* node and a *to* node. In the last model, a graph's abstract state is also a set of links and nodes. But in this model, a link consists of a *target* end and a *source* end, and each end is associated with a node. This final model allows additional state to be attached to the end of an edge. To summarize, Figure 6-8 describes the abstract state of each model.

A Superwomble user might want to collapse the portion of the model consisting of `Graph` to show only the abstract state. By considering some of these examples,

```
class Graph1 {
/* an adjacency list representation of a graph */

   Hashtable adjList;
   /* adjList contains nodes as keys, and a Vector of
      adjacent nodes as values */
}
```

```
class Graph2 {

   Vector nodes;
   Vector edges;
   /* the contents of edges are of type Edge */
}

class Edge {
   Object fromNode, toNode;
}
```

Figure 6-5: Two possible representations for a graph ADT.

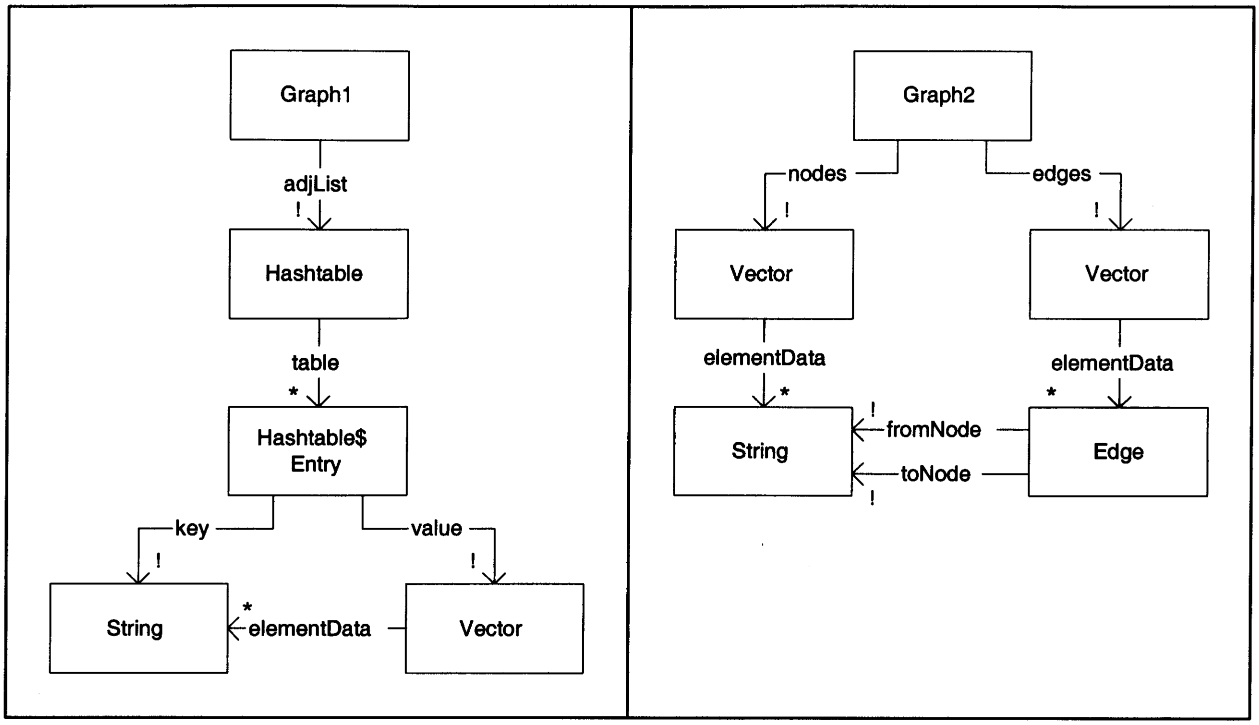

Figure 6-6: Superwomble-generated models for graph implementations.

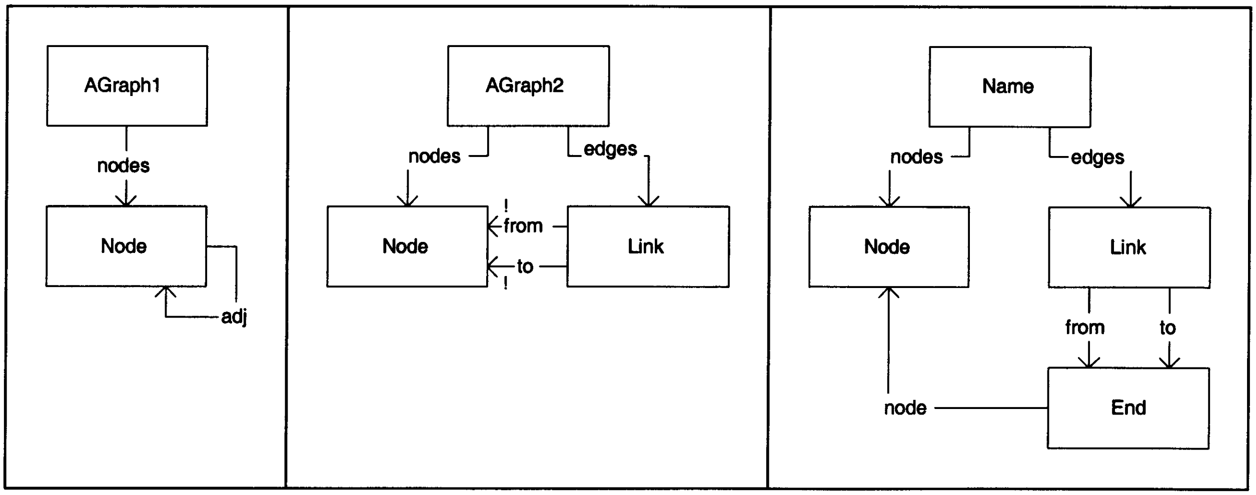

Figure 6-7: Abstract graph models.

```
Model 1:


    graph abstract state:
        nodes: set[T]
        adj: set[TxT]

Model 2:
    graph abstract state:
        nodes: set[T]
        edges: set[Link[T]]
    Link[T] abstract state:
        from: T
        to: T

Model 3:
    graph abstract state:
        nodes: set[T]
        edges: set[Link[T]]
    Link[T] abstract state:
        from: End[T]
        to: End[T]
    End[T] abstract state:
        node: T
```

Figure 6-8: Abstract state of 3 graph models

we can see some features that would be desirable in a more expressive abstraction language. We first describe the abstraction functions for the representations for each of the first two abstract models (none of these abstraction functions can be expressed in Superwomble), and based on these example abstraction functions, we discuss the features that are missing from Superwomble's language.

Consider the first representation (for `Graph1`) from Figure 6-5. If we use the first model of a graph's abstract state, its abstraction function would be:

$$\begin{aligned} AF(c) = g \mid \quad & g.nodes = c.adjList.keys() \wedge \\ & (n,m) \in g.adj \Leftrightarrow c.adjList.get(n).contains(m) \end{aligned}$$

For the second model of a graph's abstract state, the abstraction function is:

$$\begin{aligned} AF(c) = g \mid \quad & g.nodes = c.adjList.keySet() \wedge \\ & g.links = \{l \mid \quad \exists m,n : (l.from = n) \wedge (l.to = m) \wedge \\ & \qquad\qquad\qquad c.adjList.get(n).contains(m)\} \end{aligned}$$

Now consider the representation of `Graph2`. Using the first abstract state model, the abstraction function is:

$$\begin{aligned} AF(c) = g \mid \quad & g.nodes = c.nodes \wedge \\ & g.adj = \{(m,n) \mid \quad \exists e : e.fromNode = n \ \wedge e.toNode = m \ \wedge \\ & \qquad\qquad\qquad c.edges.contains(e)\} \end{aligned}$$

Using the second abstract model, the abstraction function is:

$$\begin{aligned} AF(c) = g \mid \quad & g.nodes = c.nodes \ \wedge \\ & g.edges = \{l \mid \quad \exists m,n,e : l.from = n \ \wedge l.to = m \ \wedge \\ & \qquad\qquad\qquad e.fromNode = n \ \wedge e.toNode = m \ \wedge \\ & \qquad\qquad\qquad c.edges.contains(e)\} \end{aligned}$$

Figure 6-9 shows the models that result from applying the abstraction functions to the graphs in Figure 6-6. These abstraction functions show several constructs which

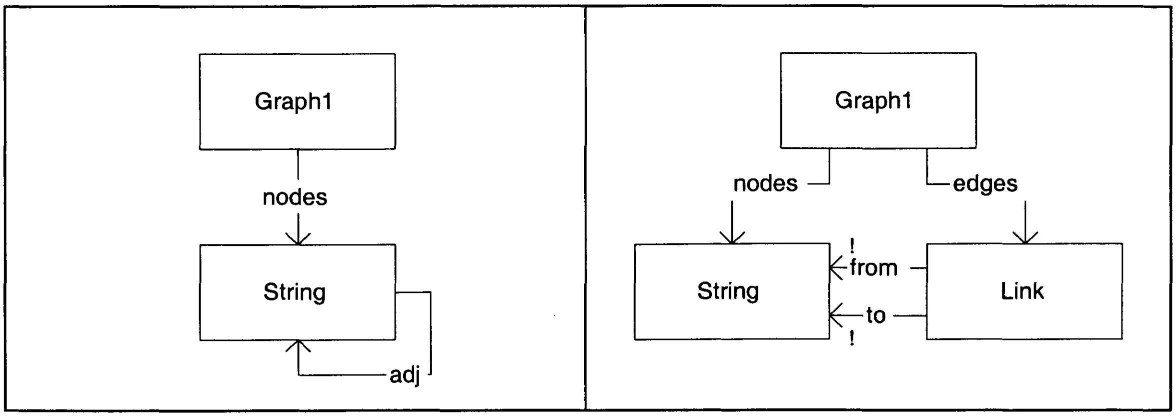

Figure 6-9: Graph models after abstraction functions have been applied.

cannot be expressed with Superwomble's abstraction language.

One common type of abstraction is to introduce abstract relations that are derived from the concrete relations (often these abstract relations replace the concrete relations entirely). Superwomble's abstraction features offer this type of abstraction in only a limited way – abstract relations must be derived by collapsing a sequence of concrete relations into one. However, in some cases, a user may want to create an abstract relation over the domains (A, B) that does not correspond to a field in A at all. The abstract model in Figure 6-9(a) is an example – *adj* does not correspond to any field in `String`. One potential feature that could be added to Superwomble's abstraction language is the ability to more expressively define derived relations.

The model in Figure 6-9(b) includes a domain that does not exist in the original object model at all. This can be useful when you want to represent an abstract notion (like a link or edge in a graph) for which there is no class in the code (perhaps for performance reasons). Also, it perhaps makes sense that if users can remove domains (when a sequence of relations are collapsed into one), they should also be able to introduce new domains. Introducing a new domain would of course be coupled with introducing new relations (since it would not be useful to introduce a domain with no relations).

Finally, users may wish to include indexed relations in their object models, which is not currently supported by Superwomble. Indexed relations are essentially ternary relations – an indexed relation $r[B]$ over $A \times C$ relates A's to C's indexed on a B. This is best explained with an example. Consider our old friend, Company. Say we wanted to keep track of employees based on some unique employee id number. We could implement this as:

```
class Company {
   Hashtable employees;
   // maps an id number (Integer) to an employee (Person)
...
}
```

We can model *employees* as an indexed relation on Integer from Company to Person. This would be modeled as:

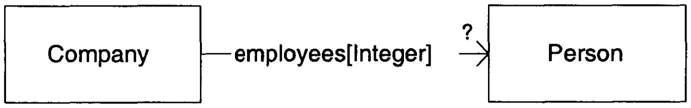

Figure 6-10: For each Integer id number, there is one Person in the employee relation of Company.

The ? on the head of the *employees* relation arrow means that for a particular index into the relation (id number), there are zero or one Persons in a single Company. That is, with any one company, id numbers are unique. One way to think of this is that the relation *employees* is actually a set of relations; a single relation can be obtained by indexing into the set with an id number. The resulting relation is a many-to-zero or one relation from Company to Person.

While a normal relation is mathematically denoted as a set of pairs, an indexed relation is denoted as a set of 3-tuples [8]. So if in company c, there is a person p with id number i, then $(c, p, i) \in$ *employees*. (An alternative interpretation, describe in [5] is to view an indexed relation on C over $A \times B$ as a relation over $C \times (A \rightarrow B)$. We will use the former interpretation in our abstraction functions.)

Superwomble does not use indexed relations in its analysis in any way. But indexed relations are a succinct way to express certain constraints graphically. For instance,

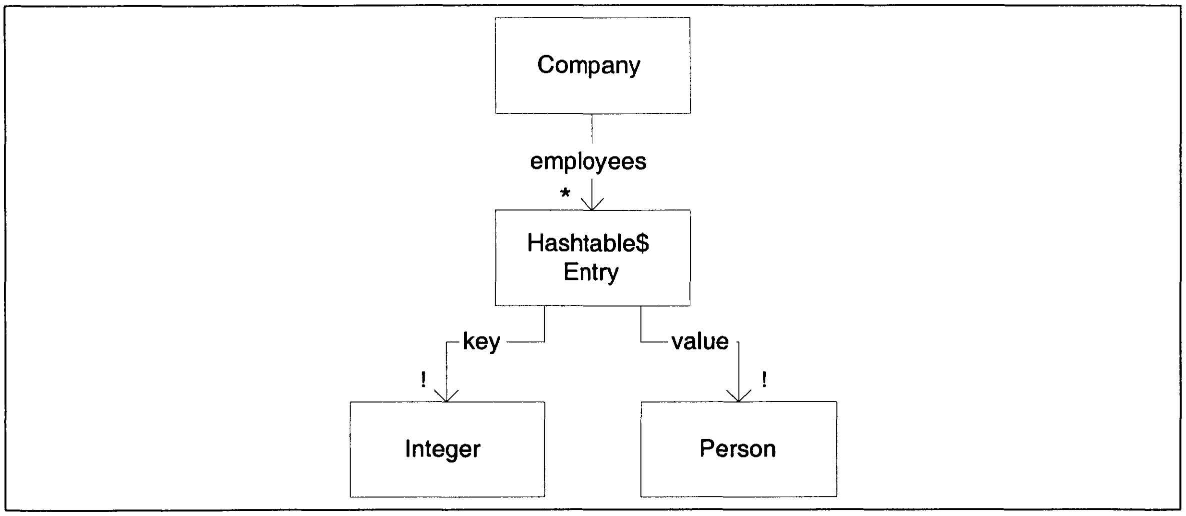

Figure 6-11: Superwomble-generated model of Company, with standard abstraction rules applied.

consider the Company example above. The model that Superwomble would generate (after the standard abstraction rules for Hashtable are applied) appears in Figure 6-11. In this model, there is no way to graphically express the fact that for id numbers are unique within a particular company. This is one case where indexed relations would be useful; not only does it allow that constraint to be expressed graphically, but it simplifies the model in other ways too (for instance, the model is less cluttered with the indexed relation syntax).

Beyond particular features of the abstraction language, it would be desirable to have a language that more closely resembles the language typically used in abstraction functions. In the current language, since the types of abstraction rules are so constrained, this is not really a problem. However, if the language was extended in the ways described, we would also want to redesign the syntax of the language, to allow things like declaring new domains and relations, and defining relations in terms of the concrete relations (fields) in the object model. This is a minor concern, since there is no standard language for abstraction functions that programmers widely use.

Chapter 7

Results

7.1 Using Superwomble Models

Superwomble models are useful for many purposes. Like its predecessor, Womble [9], some uses for the models generated by Superwomble are:

- Sketching gross structure. Superwomble can be used by a programmer unfamiliar with a program, to succinctly express the overall structure of a program.

- Comparing to design-time models. In programs where a design-time model is available, comparing the Superwomble-generated model can be useful for finding bugs.

- Identifying design patterns. An object model can often reveal the presence of a design pattern in a program. The discovery of a design pattern can help to clarify a programmer's understanding of the code.

- Identifying design anomalies. An object model can often suggest redundancies in the data structures of a program, or other places where invariants must hold.

- Detecting bugs. The multiplicity and mutability annotations are particularly helpful for finding bugs. By comparing these annotations to the expectations of the programmer, bugs can often be found. For instance, a mutable relation in a data structure that was thought to be immutable could reveal the presence of a bug to a programmer. Multiplicity markings can similarly reveal an unexpected null reference.

7.2 Case Studies

In this chapter, we apply Superwomble to several small to medium-sized programs and discuss the results. We begin with a small program, Foliotracker, and then study parts of two larger programs, CTAS and Grappa.

7.2.1 Foliotracker

Foliotracker is a program to track prices in a stock portfolio; it was developed as an example system for an undergraduate course in software engineering. Figure 7-1 shows an object model generated by Superwomble of the package containing the core data structures of Foliotracker. Foliotracker is a simple program; it would be quite easy to gain an understanding of its structure without the help of a tool. However, it demonstrates the functionality of Superwomble – how it shows polymorphism in container class, the multiplicity and mutability markings it supplies, and the abstraction rules that it provides – in a small, easy-to-understand program.

The top-level data structure in the program is `FolioTable`, which represents a set of named stock portfolios. `FolioTable` contains a mapping called `folios` from `String` to `Folio`. A `FolioTable` also contains exactly one `String` called `current_name` and exactly one `Folio` called `current`. The model suggests that there may be an invariant that there is a mapping in `folios` from `current_name` to `current`. (This invariant is in fact true, but this is impossible to tell from the object model.)

A `Folio` represents a stock portfolio – it contains a sequence of `Positions` called `positions` and may have a `value` of type `DollarAmount`. A `Position` contains exactly one `Stock`, which cannot change over time. A `Position` has a low and high price, and may contain an `Alert`. An `Alert` contains a price, a watermark, and a date when it was created. The state of an `Alert` is immutable. `Alerts` allow users to

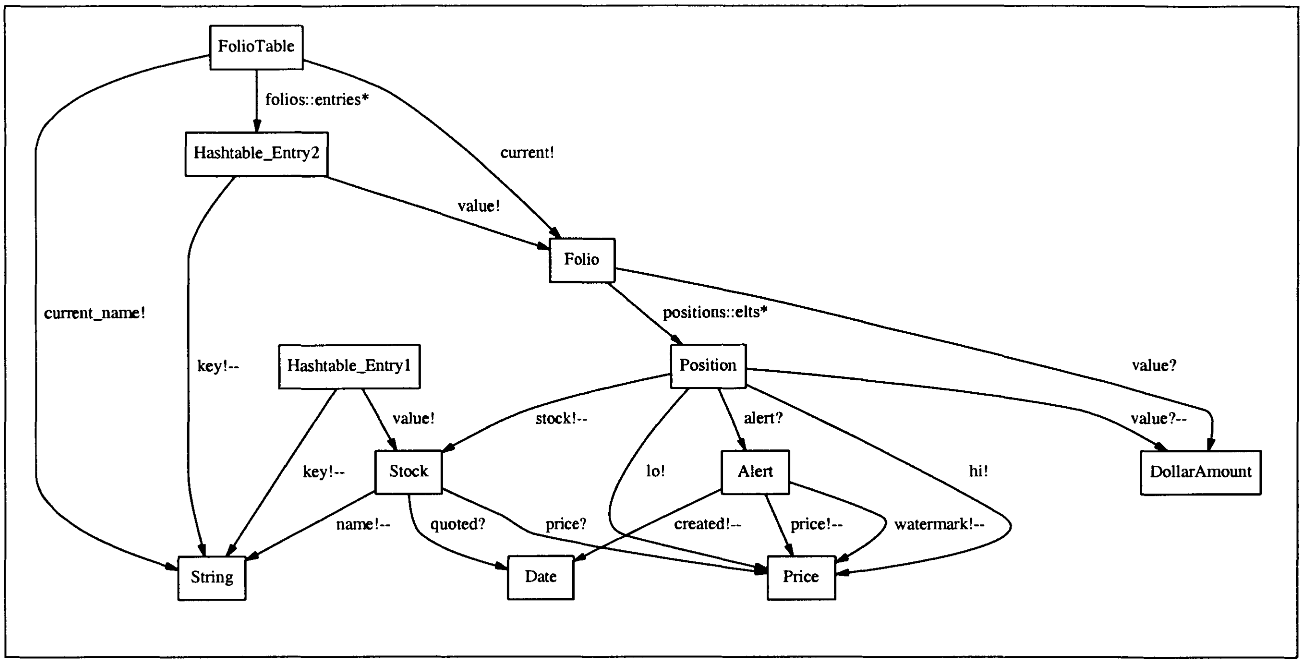

Figure 7-1: Superwomble-generated object model of Foliotracker.

be notified when the price of a stock falls below or above some specified price – its two subclasses (`LoAlert` and `HiAlert`) handle these two cases. A `Stock` has a name (which cannot change over time), and may have a price and a date on which that price was quoted.

7.2.2 CTAS

As an exercise in software engineering, students in a graduate seminar on software engineering re-engineered part of CTAS, an air-traffic control system developed by NASA [7]. Figure 7-2 shows the result of applying Superwomble to the Java reimplementation. The part of the system which was reimplemented is the communication manager, which coordinates communication between many route analyzers, which determine trajectories of aircraft, and the central aircraft database (which keeps track of aircraft location, trajectory, etc.)

The communication manager consists of `Clients` which are proxies for route analyzers running on other machines. A `Client` is associated with a `ClientGroup` which may be associated with many `Clients`. Superwomble correctly infers the mutabilities of these relations – a Client's group may not change, but new Clients may be added to a ClientGroup. Each ClientGroup has a HandlerManager, which coordinates message-handling for that group. A HandlerManager has handlers for incoming and outgoing messages; postRecvHandlers and preRecvHandlers keep track of how to handle incoming messages (the two different HandlerTables are used for prioritization), and sendHandlers keeps track of how to handle outgoing messages. Each HandlerTable relates MessageHandlers to ClientFilters, which determine which MessageHandler a message should be forwarded to, depending on the Client that the message is to or from. Finally a HandlerManager has a defaultRecvHandler, which is the default MessageHandler if no other handlers can be matched by the filters.

A MessageProcessor is associated with a ClientGroup, a RAManager and an AircraftTable. The AircraftTable maps unique AircraftIds to Aircrafts. An RAManager keeps track of assignments between route analyzers and aircraft. It may contain several Clients in the field RAs. It also contains a mapping from AircraftIds to Clients, which represents which route analyzer is handling an aircraft. An RAManager also keeps track of a set of unassigned AircraftIds. Finally, an RAManager keeps track of the load on each route analyzer in RALoads, which maps Clients to Integers.

The system also contains a ConnectionManager which handles the lower-level network communication for a ClientGroup. It consists of a Scheduler which schedules when a route analyzer should compute a trajectory.

This larger program is an example of the usefulness of Superwomble in determining the structure of data in a program. There is extensive use of polymorphic container classes, such as Vectors and Hashtables, many of which are nested containers (for instance, mainRecvHandlers maps an Integer to a mapping from Integers to MessageHandlers). Using only the program text, it would be difficult to determine what these data structures contain, without reading through much of the code, to find where, for instance, values are added to a Vector, in order to determine the types in that Vector.

This model also shows one of the weaknesses of Superwomble. Since the analysis is local (that is, only the code and method calls in a class are considered when analyzing that class), the type in socketQueues is not determined to be anything more specific than Object. In fact, socketQueues always contain Sockets. But socketQueues is only modified by code in an inner class of ConnectionManager, which is only invoked from outside of ConnectionManager.

One of the most interesting features of this model is the type shown in the field handlers in HandlerTable. The model shows that it is a mapping from Integers to a mapping from MessageHandlers to ClientFilter.OrHelpers (a subclass of ClientFilter). In reality, handlers can contain any type of ClientFilter. The type shown in the model is an artifact of the unsound type analysis – in the code of HandlerTable, both ClientFilters and ClientFilter.OrHandlers are added to the table; the two types are merged to the subtype.

Based on actual usage of HandlerTable in the rest of the program, the analysis should be able to infer the type as ClientFilter. That is because in the top-level class that runs the program (Main, which does not appear in the model), other subclasses of ClientFilter do appear as the types added to the HandlerTables in the system. However, these top-level uses of HandlerTable (or actually HandlerManager, which uses HandlerTable) are local variables in Main; since they are not fields in any class, they do not appear in the object model. This is one weakness of Superwomble; we assume that the "state" of the program is modeled by the fields of its classes. This makes sense most of the time – local variables usually represent transient relationships between objects, so they should not be considered part of the state. But at the very top-level of a program, the local variables may represent persistent state of the program; the objects created in the top-level method may certainly persist through the entire run of the program. One way to improve the way "state" is modeled by Superwomble is to allow the user to specify what the top-level method of the program is, and the local variables in that method could be considered part of the program's state.

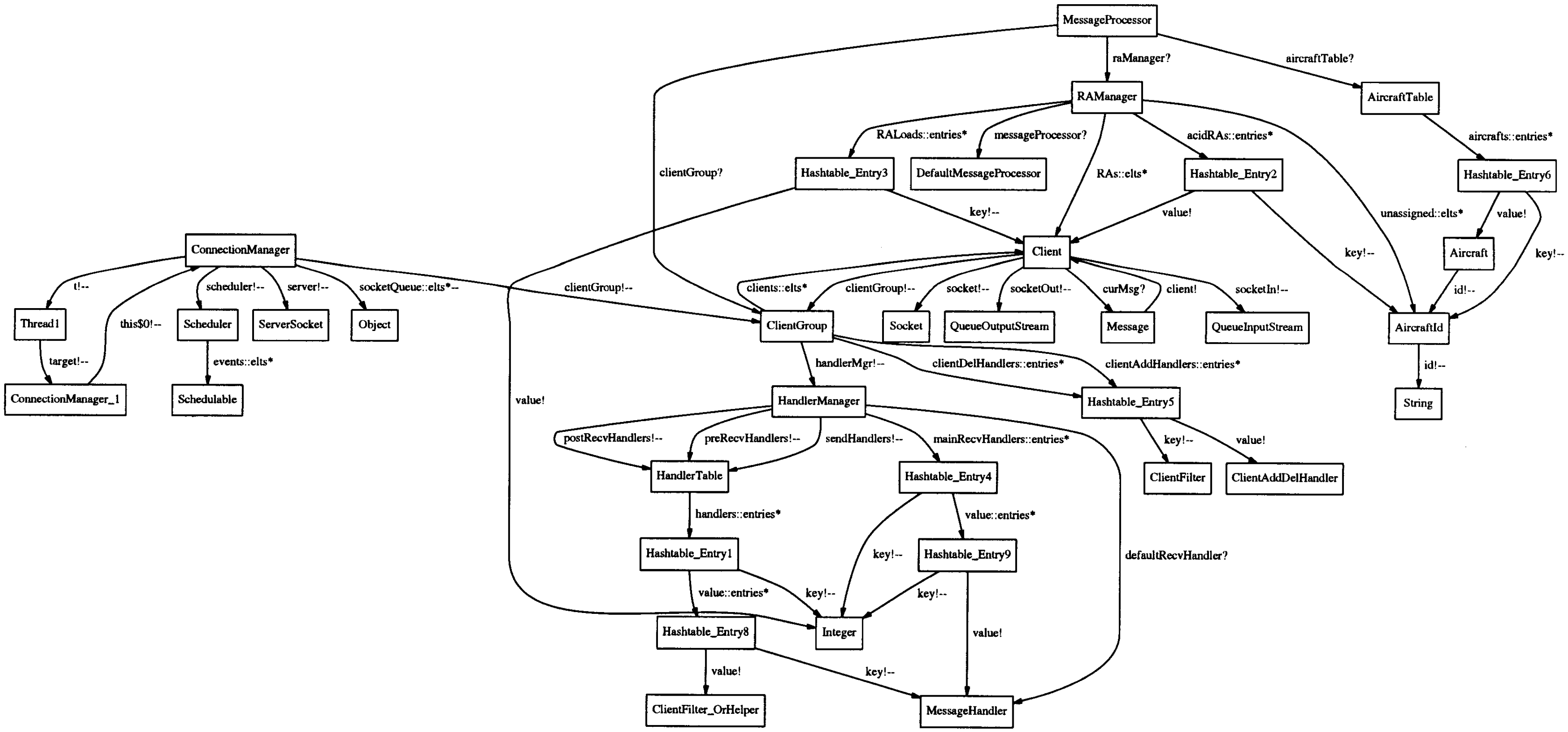

Figure 7-2: Object model of Java reimplementation of CTAS.

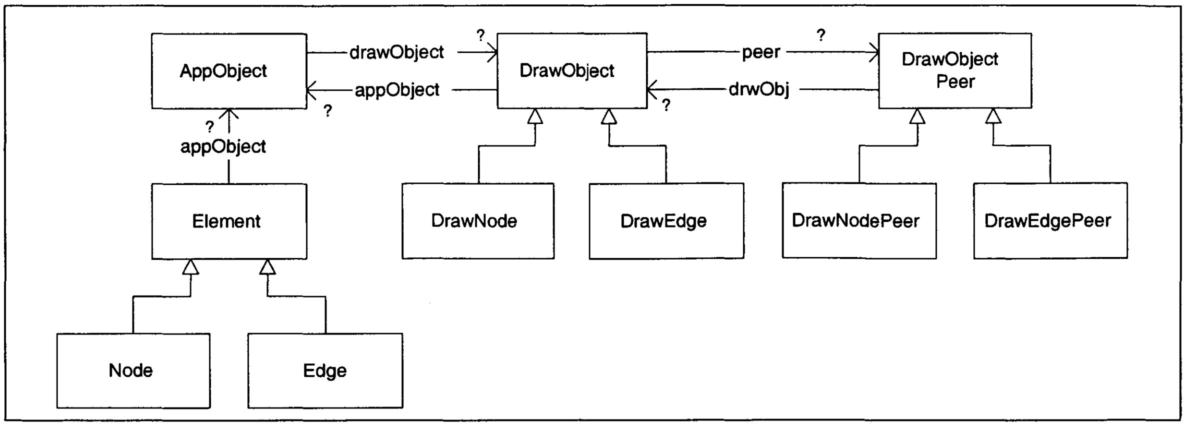

Figure 7-3: Object model of Bridge pattern in Grappa.

7.2.3 Grappa

Grappa is a Java toolkit for graph layout, created at AT&T Research. It is used by Superwomble for the graph layout; we generated an object model of Grappa when we were considering whether to use Grappa in Superwomble, to see the overall structure of the code. Figure 7-3 shows a part of the model generated for Grappa, which shows an instance of the Bridge pattern [4]. This is an example where Superwomble can allow users to identify the use of design patterns in their code. (The figure shown in Figure 7-3 is not the actual output of Superwomble; due to limitations of the graph layout program, the layout of this particular example is very hard to digest in the form output by Superwomble; the output has been laid out by hand in this example.)

7.3 Analysis

7.3.1 Performance

One of the goals of Superwomble that was stated earlier in this thesis was that it be lightweight; that is, the performance should scale well. Table 7.1 summarizes the runtime performance of Superwomble on the three programs presented in the last section, as well as some statistics on the size of the programs. The statistics were collected on a 366 MHz Intel Celeron running Windows NT with 64 MB of RAM. The largest of the 3 programs is just under 20,000 lines of code, and takes under a

program	code size (Kloc)	# of classes	analysis time (sec)
Foliotracker	1	15	6
CTAS	7.5	63	15
Grappa	17	34	43

Table 7.1: Runtime performance of Superwomble on sample programs

minute to analyze.

7.3.2 "Correctness" of Results

The case studies suggest that the unsoundness of the type analysis rarely causes problems in the output. To further quantify the unsoundness of the approach, we determined for how many edges in the object model the target types were a result of unsound merges. Table 7.2 summarizes the data for the three programs presented in the last section. Three different figures are given: the first column shows the number of edges arising from unsound merges (versus the total number of edges) before any folding of equivalent nodes occurs; the second column shows the number of edges arising from unsound merges (versus total number of edges) after equivalent nodes have been folded; the third column shows data for after abstraction rules were applied. In the latter two cases, an edge is considered to be an unsound merge if any edge that was folded into that edge is marked as unsound. For this reason, the proportion of edges that are marked unsound increases between before and after folding; the data for before folding is a better representation of how often unsound merging occurs during the analysis. Using the pre-folded data, we see that unsound merges actually do not affect the output often – on average, fewer than 5% of the edges are involved in unsound merges. Since unsound merges appear relatively infrequently, it might be useful to mark the edges that are results of unsound merges, so that the user knows that the results could be incorrect.

	unsound merged edges/total edges		
program	before node folding	after node folding	after abstraction
Foliotracker	4/61	2/39	1/18
CTAS	33/734	10/266	10/137
Grappa	80/2005	34/539	34/163

Table 7.2: Frequency of unsound merges in Superwomble

Chapter 8

Related Work

8.1 Related Type Analyses

In this section, we discuss some other applications of type analysis to program analysis tools that provide program information to programmers.

8.1.1 Soft Typing

Soft typing infers types of dynamically typed programs [12]. The aim of soft typing is two-fold – it can determine when runtime type checks are guaranteed to succeed, so that these checks can be removed (which speeds up program execution by 10 to 15% in Wright and Cartwright's *Soft Scheme*); it also flags the parts of the program where runtime checks could not be guaranteed to succeed. This warns the user of where runtime type errors may occur; however, it may report spurious type warnings, when it does not detect specific enough type information to rule out a type error. Soft Scheme also reports to users cases where the runtime type check is guarantee to fail (these are reported as errors, as opposed to warnings). Soft typing is related to Superwomble in the sense that they may both infer overly general type information to the user. However, soft typing is based on a sound type system, and will never resolve a type to something more specific than it can really be (if it did this, then it might not include necessary runtime checks). Also, soft typing uses union types which can both include and disclude types. In a language like Scheme, which lacks subtype polymorphism, merging types into some useful subtype is not really possible; and union types allows more precision which results in fewer spurious warnings reported to the user.

8.1.2 Ajax

Ajax [10] is a tool that provides a sound, static, global alias analysis of Java bytecode programs, along with an interface on which a variety of software engineering tools can be implemented [10]. One tool that has been implemented in the Ajax framework is a tool for object model extraction. The analysis, SEMI, is based on type inference with polymorphic recursion. The Ajax analysis is sound, so in and in many cases the types that appear in the object model are overly general. An example of an Ajax object model is include in the next section.

8.2 Related Tools

There are several existing tools that automatically generate object models from code. We discuss several of these tools, and compare the results of analyzing CTAS for several of them.

8.2.1 Commercial Tools

Many commercial CASE tools, such as Rational Rose and CayenneSoft ClassDesigner, can generate object models from code. These tools use simple analysis techniques which generate object models that very closely mirror the structure of the code. In Rational Rose, for instance, associations in the object model correspond exactly to the declared types of fields. There are no attempts to infer multiplicity, except in the case of arrays. There are no attempts to infer containers, or their contents. Rose also requires that all code that is used by the program be present in order to analyze the program. This increases the burden on the user, and makes it difficult to analyze

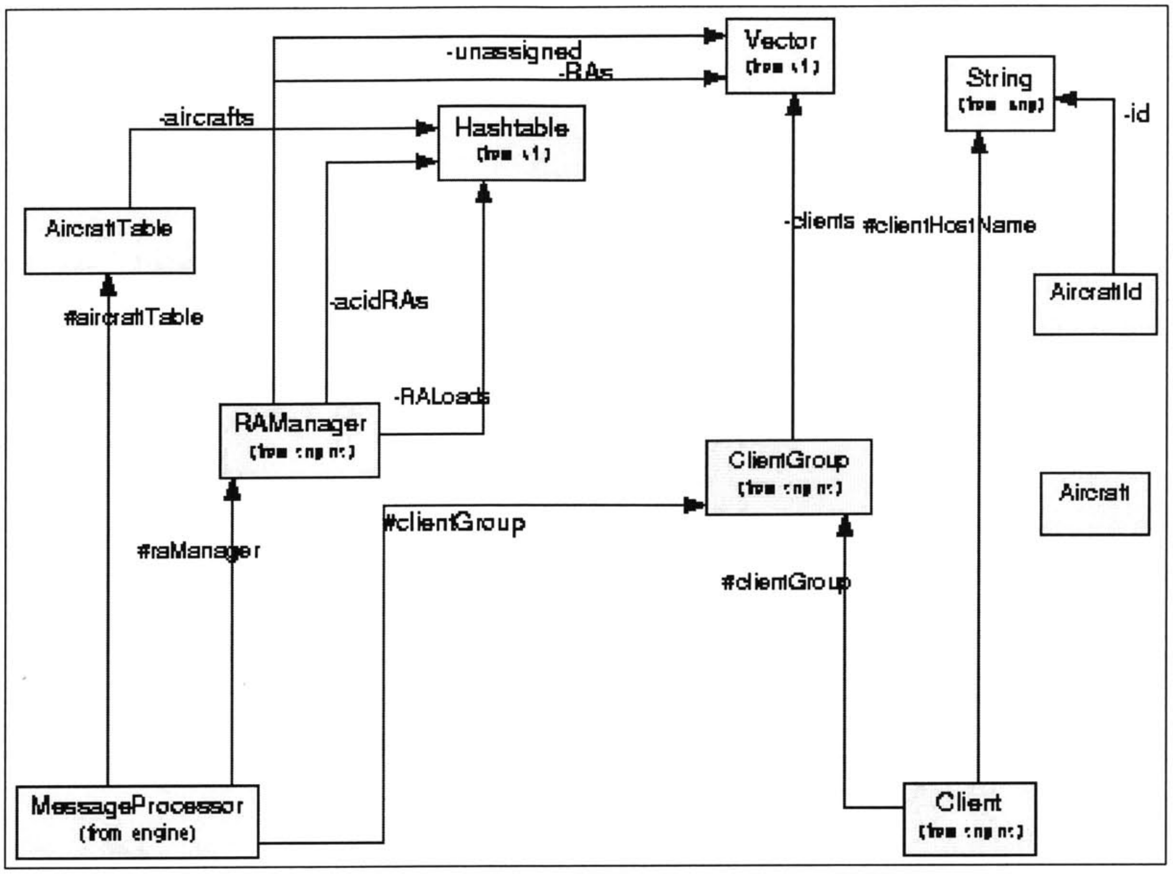

Figure 8-1: Rational Rose-generated object model of Java reimplementation of CTAS.

programs that are not complete.

Figure 8-1 shows an object model generated by Rose for part of the Java reimplementation of CTAS (for the classes *MessageProcessor, AircraftTable, RAManager, Client, ClientGroup, Aircraft,* and *AircraftID*). The types in polymorphic containers (*Vector* and *Hashtable*) are not resolved. As a result, *Aircraft* and *AircraftId* are completely disconnected from the rest of the model. It is difficult to determine what information is stored in fields like *acidRAs, RALoads, unassigned,* and *RAs.* Furthermore, the relations are not annotated with mutability or multiplicity information.

8.2.2 Womble

The predecessor of Superwomble is Womble, which uses a lightweight extraction technique that relies on heuristics in order to infer which classes are containers, the contents of containers, multiplicities, and mutabilities. In most cases, Womble can

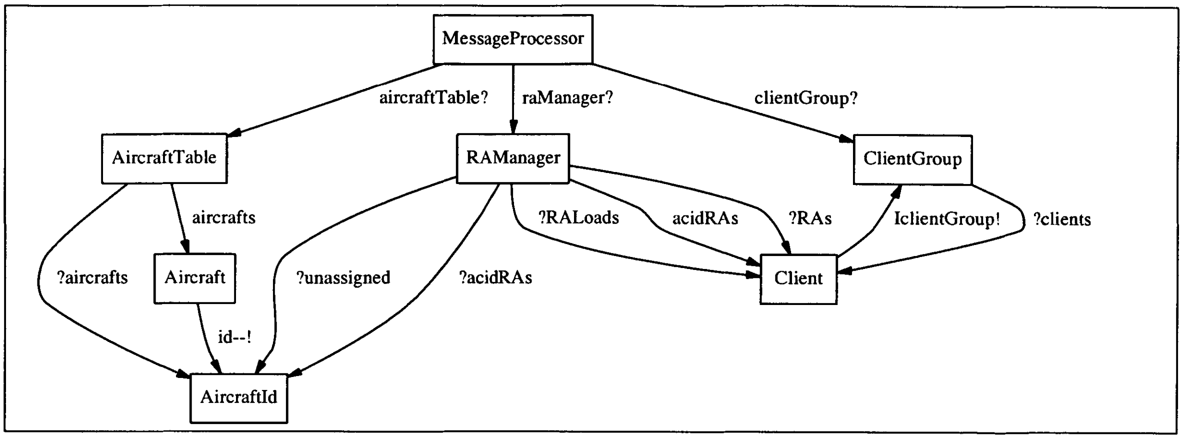

Figure 8-2: Womble-generated object model of Java reimplementation of CTAS.

correctly infer that a class is a container, even if the class is user-defined, without any input from the user. However, Womble does not handle container nesting at all. For instance, if there were a *Hashtable* mapping *Strings* to *Vectors* of *Integers*, the model would elide the *Hashtable* to show *Strings* and *Vectors*, but there would be no reference to *Integers*. In addition, Womble can sometime produce unpredictable results with respect to containers; for instance, it may infer some class to be a container, when it is not a container at all. Womble attempts to infer multiplicities on both ends of a relation, by doing a simple type escape analysis to determine the multiplicity at the tail of an edge. For an in-depth discussion of these techniques, see [9].

Figure 8-2 shows an object model of the same portion of the Java reimplementation of CTAS. This portion of the program does not contain any nested containers, so that weakness of Womble is not apparent. However, some other differences between Womble and Superwomble are evident. An *AircraftTable* contains a mapping from *AircraftId* to *Aircraft*. In this model, it merely shows relations from *AircraftTable* to both *AircraftId* and *Aircraft*. Womble would show the same sort of model if *aircrafts* were a heterogeneous *Vector* containing *Aircrafts* and *AircraftIds*.

In some specific cases, Womble can actually generate better models than Superwomble. These arise from the fact that Womble assumes all downcasts to be successful (or to at least be potentially successful), while Superwomble ignores downcasts. That is, if the type of some object is unknown, and a downcast is encountered, the Womble analysis will assign the casted type to the object. As a result, in a method where, for instance, a data structure is passed in (with the content type unknown), and the contents are cast to some type, Superwomble will not use this information to make a judgment about the type in the structure. This is technically more correct than assuming a cast can succeed; however, it gives the user less information than he could get from reading the program text.

8.2.3 Ajax

An Ajax object model of the Java reimplementation of CTAS is shown in Figure 8-3 (this figure shows a larger portion of the program than is shown in the Superwomble model). In some parts of the model, the types in polymorphic containers are not resolved to be as specific as the types resolved by Superwomble. For instance, the *key* and *value* types in *AircraftType.aircrafts* and *RAManager.acidRAs* are all shown as Object, while Superwomble finds more specific types for those fields (*AircraftID* and *aircrafts* in *AircraftTable*, and *AircraftId* and *Client* in *acidRAs*). One possible reason for this is that using the Ajax analysis, if some container class contains both *Aircraft* and *Client* objects, then all instances of those objects may end up aliased (and thus merged in the resulting object model) [2].

The model produced by Ajax is in some ways superior to a Superwomble model. For instance, Ajax can detect when fields have no values; in the CTAS model, some of the *HandlerTables* contain empty *Hashtables*. It turns out that these tables are never used in the program. Ajax performs its analysis starting from a top-level method that is defined by the user; in this case, based on the use from that top-level method, those tables are always empty. Superwomble can only infer the types that a field may take, and while its multiplicity annotations can show when a field may be empty, it cannot show that a field must be empty. Ajax's notion of top-level methods could also allow the local variables in the top-level methods to be modeled as part of the program's state, which would solve some of the problems in the Superwomble CTAS model discussed in Chapter 7 to be solved.

Analysis aside, Ajax lacks some of the useful features in Superwomble, such as abstraction [10] (but this could be done by postprocessing the data). In the model in

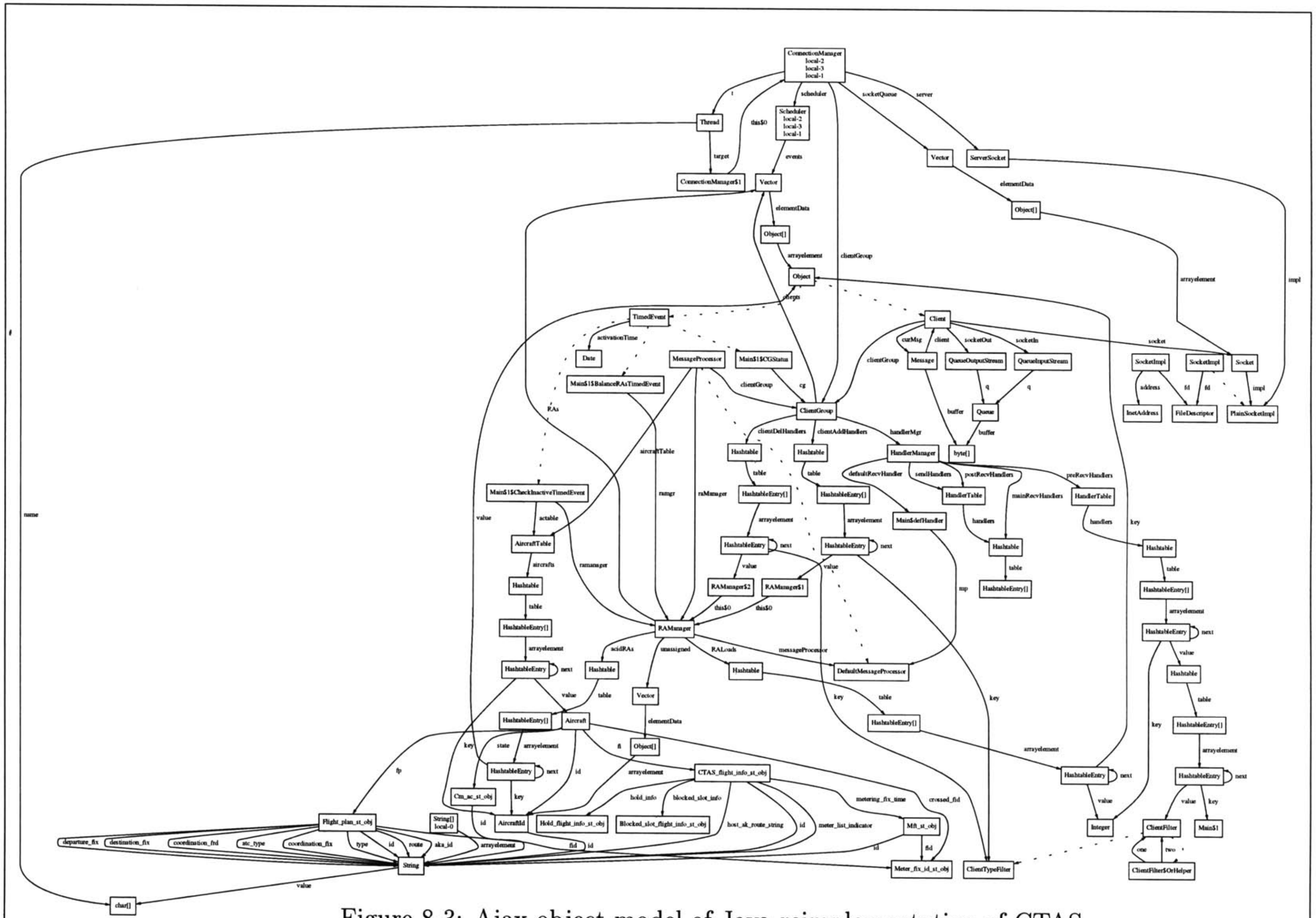

Figure 8-3: Ajax object model of Java reimplementation of CTAS.

Figure 8-3, there is no abstraction of container types, which makes the model quite large and difficult to read.

8.2.4 Other

Seeman, et al. used a pattern-based analysis for extracting object models from Java source code [11]. This technique adds an association from class `A` to class `B` to the model if `A` contains a field of type `B`, and `A` calls some method of `B`. When certain core Java container classes like `Vector` and `Hashtable` are used, the multiplicities are set to zero or more, but there is no inference of the types inside of those containers. Container nesting is not handled.

Chapter 9

Conclusions

Superwomble shows that a simple type analysis can be used to automatically generate code object models from Java bytecode. The tool is incremental, so that a programmer can analyze a partial program if some of the code is missing or not yet implemented. Concrete code object models tend to be quiet large; Superwomble's approach to solving this problem is to allow users to supply abstraction rules which can be automatically applied to the object model. The abstraction language is limited to allowing sequences of edges to be collapsed into a single edge or to be elided entirely. The most significant area for future work is to expand the abstraction language to allow the expression of more general abstraction functions.

One interesting feature of Superwomble's technique is that it relies on an unsound type analysis. The unsoundness arises from the method for merging types, and is an engineering tradeoff which makes it possible to use a flow-insensitive analysis (which improves performance) and allows analysis of incomplete programs. This is an important tradeoff in order to make the tool usable – if a tool is not fast and incremental, it greatly increases the burden on the user. In this way, Superwomble has met the goals of automatically extracting abstract object models in a lightweight and incremental way.

Superwomble is freely available at http://sdg.lcs.mit.edu/womble/.

Bibliography

[1] *Rational Rose Product Information.* Available electronically at http://www.rational.com/products/rose/prodinfo.jsp.

[2] Personal correspondence with Robert O'Callahan (author of Ajax), May 2001.

[3] Course materials for 6.170 (laboratory in software engineering). Available electronically at http://web/6.170/www/psets/ps5/ps5.html, Fall 2000/Spring 2001.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Reading, Mass., 1995.

[5] Daniel Jackson. Alloy: A lightweight object modelling notation. Technical Report MIT-LCS-797, MIT Laboratory for Computer Science, Cambridge, Mass., February 2000.

[6] Daniel Jackson. Lecture notes for 6.170 (laboratory in software engineering). Available electronically at http://web/6.170/www/Old-Fall00/handouts/lectures/, Fall 2000.

[7] Daniel Jackson and John Chapin. Redesigning air-traffic control: A case study in software design. *IEEE Software*, May/June 2000.

[8] Daniel Jackson and Ilya Shlyakhter. A micromodularity mechanism. To appear, Foundations in Software Engineering.

[9] Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. *IEEE Transactions on Software Engineering*, February 2001.

[10] Robert O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools.* PhD thesis, Carnegie Mellon University, School of Computer Science, 2001.

[11] Jochen Seemann and Juergen Wolff von Gudenberg. Pattern-based design recovery of java software. In *Proc. Foundations of Software Engineering*, November 1998.

[12] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. *ACM Transactions on Programming Languages and Systems*, January 1997.