Understanding Class Hierarchies Using Concept Analysis

GREGOR SNELTING
Universität Passau
and
FRANK TIP
IBM T.J. Watson Research Center

A new method is presented for analyzing and reengineering class hierarchies. In our approach, a class hierarchy is processed along with a set of applications that use it, and a fine-grained analysis of the access and subtype relationships between objects, variables, and class members is performed. The result of this analysis is again a class hierarchy, which is guaranteed to be behaviorally equivalent to the original hierarchy, but in which each object only contains the members that are required. Our method is semantically well-founded in *concept analysis*: the new class hierarchy is a minimal and maximally factorized *concept lattice* that reflects the access and subtype relationships between variables, objects and class members. The method is primarily intended as a tool for finding imperfections in the design of class hierarchies, and can be used as the basis for tools that largely automate the process of reengineering such hierarchies. The method can also be used as a space-optimizing source-to-source transformation that removes redundant fields from objects. A prototype implementation for Java has been constructed, and used to conduct several case studies. Our results demonstrate that the method can provide valuable insights into the usage of a class hierarchy in a specific context, and lead to useful restructuring proposals.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement—*restructuring, reverse engineering, reengineering*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and Objects, Inheritance*; F.3.2 [**Logics and meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms: Algorithms, Documentation, Experimentation, Languages, Theory

Additional Key Words and Phrases: Class hierarchy reengineering, concept analysis

1. INTRODUCTION

Designing a class hierarchy is hard, because it is not always possible to anticipate how a hierarchy will be used by an application. This is especially the case when

A preliminary version of parts of this article appeared in the Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1998 [Snelting and Tip 1998]. This work is supported by Deutsche Forschungsgemeinschaft, grant Sn11/7-1.
Authors' addresses: G. Snelting, Fakultät für Mathematik und Informatik, Universität Passau, Innstr. 33, 94032 Passau, Germany; F. Tip, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

```
class String { /* details omitted */ };
class Address { /* details omitted */ };
enum Faculty { Mathematics, CompScience };
class Professor; /* forward declaration */

class Person {
public:
  String name;
  Address address;
  long socialSecurityNumber;
};

class Student :  public Person {
public:
  Student(String sn, Address sa, int si){
    name = sn; address = sa; studentId = si;
  };
  void setAdvisor(Professor *p){
    advisor = p;
  };
  long studentId;
  Professor *advisor;
};
class Professor :  public Person {
public:
  Professor(String n, Faculty f, Address wa){
    name = n; faculty = f;
    workAddress = wa;
    assistant = 0; /* default:  no assistant */
  };
  void hireAssistant (Student *s){
    assistant = s;
  };
  Faculty faculty;
  Address workAddress;
  Student *assistant; /* either 0 or 1 assistants */
};
```

(a)

```
int main(){
  String s1name, p1name;
  Address s1addr, p1addr;
  Student* s1 =            /* Student1 */
    new Student(s1name,s1addr,12345678);
  Professor *p1 =          /* Professor1 */
    new Professor(p1name,Mathematics,p1addr);
  s1->setAdvisor(p1);
  return 0;
}
```

(b)

```
int main(){
  String s2name, p2name;
  Address s2addr, p2addr;
  Student* s2 =            /* Student2 */
    new Student(s2name,s2addr,87654321);
  Professor *p2 =          /* Professor2 */
    new Professor(p2name, CompScience, p2addr);
  p2->hireAssistant(s2);
  return 0;
}
```

(c)

Fig. 1. Example: relationships between students and professors. **(a)** Class hierarchy for expressing associations between students and professors. **(b)** Example program using the class hierarchy of Figure 1(a). **(c)** Another example program using the class hierarchy of Figure 1(a).

a class hierarchy is developed as a library, and designed independently from the applications that use it. Ongoing maintenance, in particular ad hoc extensions of the hierarchy, will further increase the system's entropy. As typical examples of inconsistencies that may arise, one might think of:

—A class C may contain a member m not accessed in any C-instance, an indication that m may be removed, or moved into a derived class.

—Different instances of a given class C may access different subsets of C's members, an indication that it might be appropriate to split C into multiple classes.

In this article, we present a method for analyzing the usage of a class hierarchy based on *concept analysis* [Wille 1982]. Our approach comprises the following steps. First, a table is constructed that precisely reflects the usage of a class hierarchy. In particular, the table makes explicit relationships between the types of variables and class members such as "the type of x must be a base class of the type of y" and "member m must occur in a base class of the type of variable x." From the table, a *concept lattice* is derived, which factors out information that variables or members

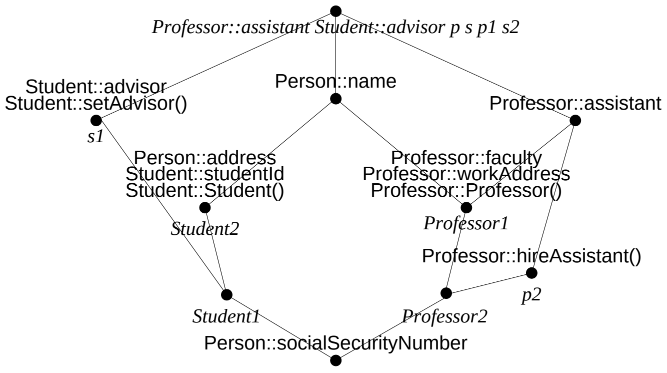


Fig. 2. Lattice for Student/Professor example.

have in common. We will show how the concept lattice can provide valuable insight into the design of a class hierarchy, and how it can serve as a basis for automated or interactive restructuring tools for class hierarchies. The examples presented in this article are written in C++ or Java, but our approach is applicable to other object-oriented languages as well.

Our method can analyze a class hierarchy along with any number of programs that use it, and provide the user with either a combined view reflecting the usage of the hierarchy by the entire set of programs, or with individual views that clarify how each application uses the hierarchy. Analyzing a class hierarchy without any accompanying applications (such as a class library) is also possible, and can be useful to study the internal dependences inside class definitions.

1.1 A Motivating Example

Consider the example of Figure 1, which is concerned with relationships between students and professors. Figure 1(a) shows a class hierarchy, in which a class `Person` is defined that contains a person's `name`, `address`, and `socialSecurityNumber`. Classes `Student` and `Professor` are derived from `Person`. `Student`s have an identification number (`studentId`), and a thesis `advisor` if they are graduate students. A constructor is provided for initializing `Student`s, and a method `setAdvisor` for designating a `Professor` as an advisor. `Professor`s have a `faculty` and a `workAddress`, and a professor may hire a student as a teaching `assistant`. A constructor is provided for initialization, and a method `hireAssistant` for hiring a `Student` as an assistant. Details for classes `Address` and `String` are not provided; in the subsequent analysis these classes will be treated as “atomic” types, and we will not attempt to analyze them.

Figures 1(b) and (c) show two programs that use the class hierarchy of Figure 1(a). In the first program, a student and a professor are created, and the professor is made the student's advisor. The second program creates another student and professor, and here the student is made the professor's assistant. The example is certainly not perfect C++ code, but looks reasonable enough at first glance.

Figure 2 shows the lattice computed by our method for the class hierarchy and the two example programs of Figure 1. Ignoring a number of details, the lattice may be interpreted as follows:

—The lattice elements (concepts) may be viewed as *classes* of a restructured class hierarchy that precisely reflects the usage of the original class hierarchy by the client programs.

—The ordering between lattice elements may be viewed as *inheritance* relationships in the restructured class hierarchy.

—A variable v has type C in the restructured class hierarchy if v occurs immediately *below* concept C in the lattice.

—A member m occurs in class C if m appears *directly above* concept C in the lattice.

Examining the lattice of Figure 2 according to this interpretation reveals the following interesting facts[1]:

—Data member `Person::socialSecurityNumber` is never accessed, because no variable appears below it. This illustrates situations where subclassing is used to inherit the functionality of a class, but where some of that functionality is not used.

—Data member `Person::address` is only used by students, and not by professors (for professors, the data member `Professor::workAddress` is used instead, perhaps because their home address is confidential information). This illustrates a situation where the member of a base class is used in some, but not all derived classes.

—No members are accessed from parameters `s` and `p`, and from data members `advisor` and `assistant`. This is due to the fact that no operations are performed on a student's advisor, or on a professor's assistant. Such situations are typical of redundant, incomplete, or erroneous code and should be examined closely.

—The analyzed programs create professors who hire assistants (`Professor2`), and professors who do not hire assistants (`Professor1`). This can be seen from the fact that method `Professor::hireAssistant()` appears above the concept labeled `Professor2`, but not above the concept labeled `Professor1`.

—There are students with advisors (`Student1`) and students without advisors (`Student2`). This can be seen from the fact that `Student::setAdvisor` appears above the concept labeled `Student1`, but not above the concept labeled `Student2`.

—Class `Student`'s constructor does not initialize the `advisor` data member. This can be seen from the fact that data member `Student::advisor` does not appear above method `Student::Student()` in the lattice.[2]

One can easily imagine how the above information might be used as the basis for restructuring the class hierarchy. One possibility would be for a tool to automatically generate restructured source code from the information provided by the

[1]The labels `Student1`, `Professor1`, `Student2`, and `Professor2` that appear in the lattice represent the types of the heap objects created by the example programs at various program points (indicated in Figures 1(b) and (c) using comments).
[2]`Student::Student()` also represents the `this`-pointer of the method.

lattice, similar to the approach taken in Tip and Sweeney [1997; 2000]. However, from a redesign perspective, we believe that an interactive approach would be more appropriate. For example, the programmer doing the restructuring job may decide that the data member `socialSecurityNumber` should be retained in the class hierarchy because it may be needed later. In the interactive tool we envision, one could indicate this by *moving up* in the lattice the attribute under consideration, `socialSecurityNumber`. The reengineer may also decide that certain fine distinctions in the lattice are unnecessary. For example, one may decide that it is not necessary to distinguish between professors that hire assistants, and professors that do not. In an interactive tool, this distinction could be removed by *merging* the concepts for `Professor1` and `Professor2`.

Another useful capability of an interactive tool would be to associate names with lattice elements. When the programmer is done manipulating the lattice, these names could be used as class names in the restructured hierarchy when the restructured source code is generated. For example, using the information provided by the lattice, the programmer may determine that `Student` objects on which the `setAdvisor` method is invoked are graduate students, whereas `Student` objects on which this method is not called are undergraduates. Consequently, he may decide to associate the names `Student` and `GraduateStudent` with the concepts labeled `Student2` and `Student1`, respectively.

1.2 Organization of this Article

The remainder of this article is organized as follows. Section 2 briefly reviews the relevant parts of the theory of concept analysis. In Section 3 we define the objects and attributes in our domain, which correspond to the rows and columns of the tables. The process of constructing tables is presented in Section 4, while Section 5 discusses important properties of the lattice, in particular behavioral equivalence. Section 6 presents extensions for constructs such as type casts. In Section 7, we discuss how the information provided by the lattice can reveal problems in the design of class hierarchies, and how the lattice can be used as a basis for interactive restructuring tools. Section 8 describes our prototype implementation for Java in some detail. Section 9 discusses several case studies. Section 10 discusses related work. Finally, conclusions and directions for future work are presented in Section 11.

2. CONCEPT ANALYSIS

Concept analysis provides a way to identify groupings of *objects* that have common *attributes*. The mathematical foundation was laid by Birkhoff in 1940 [Birkhoff 1940]. Birkhoff proved that for every binary relation between certain objects and attributes, a lattice can be constructed that provides remarkable insight into the structure of the original relation. The lattice can always be transformed back to the original relation; hence concept analysis is similar in spirit to Fourier analysis.[3]

Later, Wille and Ganter elaborated Birkhoff's result and transformed it into a data analysis method [Wille 1982; Ganter and Wille 1999]. Since then, it has found a variety of applications, including analysis of software structures [Krone

[3]A function and its Fourier transform are very different representations of the same information, but can be transformed into each other.

	small	medium	large	near	far	moon	no moon
Mercury	×			×			×
Venus	×			×			×
Earth	×			×		×	
Mars	×			×		×	
Jupiter			×		×	×	
Saturn			×		×	×	
Uranus		×			×	×	
Neptune		×			×	×	
Pluto	×				×	×	

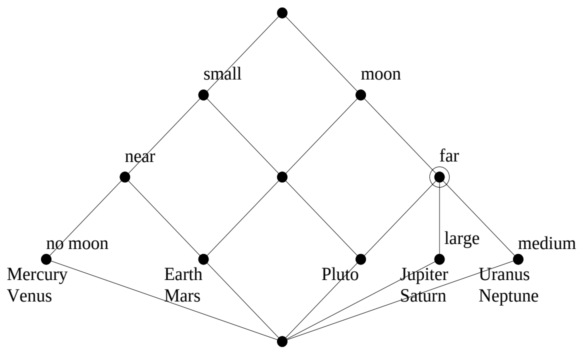


Fig. 3. Example table and associated concept lattice.

and Snelting 1994; Snelting 1996; 1998; Lindig and Snelting 1997; Siff and Reps 1997; Godin and Mili 1993; Godin et al. 1998; Ball 1999].

2.1 Relations and Their Lattices

Concept analysis starts with a relation, or boolean table, T between a set of *objects* $\mathcal{O}$ and a set of *attributes* $\mathcal{A}$; hence $T \subseteq \mathcal{O} \times \mathcal{A}$.

For any set of objects $O \subseteq \mathcal{O}$, their set of common attributes is defined as

$$\sigma(O) = \{a \in \mathcal{A} \mid \forall o \in O : (o, a) \in T\}.$$

For any set of attributes $A \subseteq \mathcal{A}$, their set of common objects is

$$\tau(A) = \{o \in \mathcal{O} \mid \forall a \in A : (o, a) \in T\}.$$

A pair (O, A) is called a *concept* if

$$A = \sigma(O) \text{ and } O = \tau(A).$$

Informally, such a concept corresponds to a *maximal rectangle* in the table T: any $o \in O$ has all attributes in A, and all attributes $a \in A$ fit to all objects in O. It is important to note that concepts are invariant against row or column permutations in the table. The set of all concepts of a given table forms a partial order via

$$(O_1, A_1) \le (O_2, A_2) \iff O_1 \subseteq O_2 \iff A_1 \supseteq A_2.$$

Birkhoff proved that the set of concepts constitutes a complete lattice, the *concept lattice* $\mathcal{L}(T)$. For two elements (O_1, A_1) and (O_2, A_2) in the concept lattice, their infimum or *meet* is defined as

$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2)),$$

and their supremum or *join* as

$$(O_1, A_1) \vee (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2).$$

A concept $c = (O, A)$ has *extent* $ext(c) = O$ and *intent* $int(c) = A$. In our figures, a lattice element (concept) c is labeled with attribute $a \in \mathcal{A}$, if it is the *largest* concept with a in its intent, and it is labeled with an object $o \in \mathcal{O}$, if it is the *smallest* concept with o in its extent. The (unique) lattice element labeled with a is denoted $\mu(a)$, and the (unique) lattice element labeled with o is denoted $\gamma(o)$. Thus

$$\mu(a) = \bigvee\{c \in \mathcal{L}(T) \mid a \in \mathit{int}(c)\}, \quad \gamma(o) = \bigwedge\{c \in \mathcal{L}(T) \mid o \in \mathit{ext}(c)\}.$$

The following fundamental property establishes the connection between a table and its lattice, and shows that they can be reconstructed from each other:

$$(o, a) \in T \iff \gamma(o) \leq \mu(a)$$

Hence, the attributes of object o are those which appear *above* o, and all objects that appear *below* a have attribute a. Consequently, join points (suprema) in the lattice indicate that certain objects have attributes in common, while meet points (infima) show that certain attributes fit to common objects. In other words, join points factor out common attributes, while meet points factor out common objects. Thus, the lattice uncovers a hierarchy of conceptional clusters that was implicit in the original table.

Figure 3 shows a table and its lattice (taken from Davey and Priestley [1990]). The element labeled *far* corresponds to the maximal rectangle indicated in the table. This element is the supremum of all elements with *far* in their intent: *Pluto*, *Jupiter*, *Saturn*, *Uranus*, *Neptune* are below *far* in the lattice, and the table confirms that these (and no other) planets are indeed far away.

2.2 Implications

A table and its lattice are alternate views on the same information, serving different purposes and providing different insights. There is yet another view: a set of *implications*. Let $A, B \subseteq \mathcal{A}$ be two sets of attributes. We say that A *implies* B, iff any object with the attributes in A also has the attributes in B:

$$A \to B \iff \forall o \in \mathcal{O} : \big(\forall a \in A : (o, a) \in T\big) \Rightarrow \big(\forall b \in B : (o, b) \in T\big)$$

For $B = \{b_1, \ldots, b_k\}$, $A \to B$ holds iff $A \to b_i$ for all $b_i \in B$.[4] Implications show up in the lattice as follows: $A \to b$ holds iff $\bigwedge\{\mu(a) \mid a \in A\} \leq \mu(b)$. Informally, implications between attributes can be found along upward paths in the lattice. In the example of Figure 3, we have that $\mu(\text{far}) \leq \mu(\text{moon})$, which can be read as $\mathit{far} \to \mathit{moon}$, or "A planet which is far away has a moon." Other examples of implication are $\mathit{nomoon} \to \mathit{near}, \mathit{small}$; or $\mathit{near}, \mathit{far} \to \mathit{large}$ (the latter implication being true because its premise is contradictory).

There is a minimal set of implications, from which all other valid implications can be derived: the *implication base* $\mathcal{I}(T)$. For the example, it consists of 10 implications, including $\mathit{far} \to \mathit{moon}$ and $\mathit{no\ moon} \to \mathit{near}, \mathit{small}$. Nonbase implications such as $far, \mathit{small} \to \mathit{moon}, \mathit{small}$ or $\mathit{no\ moon} \to \mathit{near}$ can be derived by propositional logic.

Often, some implications are known to hold *a priori*. Such background knowledge can easily be integrated into a given table. An implication $x \to y$ can be enforced by copying the entries from the x column to the y column, and will cause $\mu(x) \leq \mu(y)$

[4]We will usually write $a_1, \ldots, a_n \to b_1, \ldots, b_m$ instead of $\{a_1, \ldots, a_n\} \to \{b_1, \ldots, b_m\}$.

in $\mathcal{L}(T)$. A general implication $A \rightarrow B$ can be enforced by copying the intersection of the A columns to all B columns.

2.3 Lattice Construction

The table T, the lattice $\mathcal{L}(T)$, and the implication basis $\mathcal{I}(T)$ represent very different views onto the same information, but can be transformed into each other; furthermore, background knowledge, given as a set of implications, may be added. In this section, we will present a short description of the most important transformation: the computation of the concept lattice for a given table.

Ganter's algorithm for lattice construction utilizes the fact that $C = \sigma \circ \tau$ (as well as $C' = \tau \circ \sigma$) is a closure operator on $2^{\mathcal{O}}$: it is extensive ($O \subseteq C(O)$), idempotent ($C(C(O)) = C(O)$), and monotone ($O \subseteq O' \Rightarrow C(O) \subseteq C(O')$). $C(O)$ determines the largest object set with the same common attributes as O. It turns out that the lattice elements' extents are precisely the closed sets under C. If we have computed all the extents (that is, computed the closure system $\{C(O) \mid O \subseteq \mathcal{O}\}$), the corresponding intents are determined using σ, and the lattice, together with its partial order as defined above, is complete.

Ganter's algorithm requires that $2^{\mathcal{O}}$ is totally ordered (e.g., by numbering the objects and using the lexicographical order for object sets). The algorithm enumerates object sets according to the lexicographical order, and applies C. The process starts with $C(\emptyset)$, which determines the extent of the bottom element. Once an extent has been found, its lexicographical successors are enumerated and C is applied, until the next extent (in lexicographic order) is found.

Construction of concept lattices and implication bases has typical time complexity $O(n^3)$ for an $n \times n$ table, but can be exponential in the worst case. Empirical studies show that even for large tables, exponential behavior is extremely rare [Snelting 1996]. In fact, it can be shown that if the number of attributes for every object is bounded (which is true for most applications, including the one in the current article), the lattice size is linear in the number of table entries [Godin et al. 1998]. In practice, Ganter's algorithm needs less than a second for 2000-element lattices on a standard workstation [Snelting 1996].

Generation of the minimal implication base has the same complexity as generation of the lattice, and the number of base implications is of the same order of magnitude as the number of lattice elements.

If a row or column is added to a table, the lattice for the original table is a sublattice of the lattice for the extended table, and the new lattice can be constructed incrementally from the old one. The minimal implication base can be constructed in an incremental manner as well [Ganter and Wille 1999].

There is much more to say about concept lattices, their structure theory, and related algorithms and methodology. Davey and Priestley's book [Davey and Priestley 1990] contains a chapter on elementary concept analysis. Ganter and Wille [Ganter and Wille 1999] treat the topic in depth.

3. OBJECTS AND ATTRIBUTES

Roughly speaking, the objects and attributes in our domain are variables and class members, respectively, and the table that will be constructed in Section 4 identifies for each variable which members must be included in its type. Before we can define the objects and attributes more precisely, we need to introduce some terminology. In what follows, $\mathcal{P}$ denotes a program containing a class hierarchy, or a collection of programs that share a class hierarchy. Further, v, w, ... denote the variables in $\mathcal{P}$ whose type is a class, and p, q, ... the variables in $\mathcal{P}$ whose type is a pointer to a class (references can be treated similarly, and we omit their formalization in the present article). Expressions are denoted by x, y, ... We will henceforth use "variables" to refer to variables as well as parameters. In the definitions that follow, $\mathit{TypeOf}(\mathcal{P}, x)$ denotes the type of expression x in $\mathcal{P}$. In this article, we will assume that each expression has a single static type. Accommodating C-style unions and generic types will be discussed in Section 11.1.

The *objects* of our domain are the program variables through which the class hierarchy is accessed. Variables whose type is (pointer to) built-in can be ignored because the class hierarchy can only be accessed through variables whose type is *class-related* (i.e., variables whose type is a class, or a pointer to a class). Definition 1 below defines sets of variables *ClassVars* and *ClassPtrVars* whose type is a class, and a pointer to a class, respectively. In Section 6.1, we will discuss how to model heap-allocated objects. Note that *ClassPtrVars* includes implicitly declared `this` pointers of methods. In order to distinguish between `this` pointers of different methods, we will henceforth refer to the `this` pointer of method `A::f()` by the fully qualified name of its method, i.e., `A::f`.

Definition 1. Let $\mathcal{P}$ be a program. Then, the set of class-typed variables and the set of pointer-to-class-typed variables are defined as follows:

$$\begin{aligned}&\mathit{ClassVars}(\mathcal{P}) \triangleq \\ &\quad \{\, v \mid v \text{ is a variable in } \mathcal{P},\ \mathit{TypeOf}(\mathcal{P}, v) = C,\ \text{ for some class } C \text{ in } \mathcal{P} \,\} \\ &\mathit{ClassPtrVars}(\mathcal{P}) \triangleq \\ &\quad \{\, p \mid p \text{ is a variable in } \mathcal{P},\ \mathit{TypeOf}(\mathcal{P}, *p) = C,\ \text{ for some class } C \text{ in } \mathcal{P} \,\}\end{aligned}$$

The *attributes* of our domain are class members. Following the definitions of Tip and Sweeney [1997; 2000], we will distinguish between *definitions* and *declarations* of members. We define these terms as follows. The definition of a member comprises a member's signature (interface) as well as the executable code in its body, whereas the declaration of a member only represents its signature. This distinction is needed for accurately modeling virtual method calls. Consider a call to a virtual method f from a *pointer* p. In this case, only the declaration of f needs to be contained in p's type in order to be able to invoke f; the body of f does not need to be statically visible to p.[5] Naturally, a *definition* of f must be visible to the object that p points to at run-time, so that the dynamic dispatch can be executed correctly.

Definition 2 (shown below) defines sets $\mathit{MemberDcls}(\mathcal{P})$ and $\mathit{MemberDefs}(\mathcal{P})$ of member declarations and member definitions in $\mathcal{P}$. We distinguish between declarations and definitions of virtual methods for the reasons stated above. For nonvirtual methods, making this distinction is not necessary because the full definition of a nonvirtual method must always be statically visible to the caller. Therefore, non-

[5]Our objective is to identify the smallest possible set of member declarations and definitions that must be included in the type of any variable. Including the *definition* of f in $*p$'s type may lead to the incorporation of members that are otherwise not needed (in particular, members accessed from f's `this` pointer).

```
class A {
public:
  virtual int f(){ return g(); };
  virtual int g(){ return x; };
  int x;
};
class B : public A {
public:
  virtual int g(){ return y; };
  int y;
};
class C : public B {
public:
  virtual int f(){ return g() + z; };
  int z;
};
```

```
int main(){
  A a; B b; C c;
  A *ap;
  if (...)  { ap = &a; }
  else { if (...)  { ap = &b; }
         else { ap = &c; } }
  ap->f();
  return 0;
}
```

Fig. 4. Example program $\mathcal{P}_1$.

virtual methods are modeled using definitions only. Data members are modeled as declarations because they have no `this` pointer from which other members can be accessed.

Definition 2. Let $\mathcal{P}$ be a program. Then, we define the set of member declarations and member definitions as follows:

$$\begin{array}{l} \mathit{MemberDcls}(\mathcal{P}) \triangleq \\ \quad \{\ \mathit{dcl}(C{::}m)\ |\text{m is a data member or virtual method in class C}\ \} \\ \mathit{MemberDefs}(\mathcal{P}) \triangleq \\ \quad \{\ \mathit{def}(C{::}m)\ |\text{m is a virtual or nonvirtual method in class C}\ \} \end{array}$$

Example. Figure 4 shows a program $\mathcal{P}_1$ that will be used as a running example. For $\mathcal{P}_1$, we have

$$\begin{array}{lcl} \mathit{ClassVars}(\mathcal{P}_1) & \equiv & \{\ \mathtt{a},\ \mathtt{b},\ \mathtt{c}\ \} \\ \mathit{ClassPtrVars}(\mathcal{P}_1) & \equiv & \{\ \mathtt{ap},\ \mathtt{A::f},\ \mathtt{A::g},\ \mathtt{B::g},\ \mathtt{C::f}\ \} \\ \mathit{MemberDcls}(\mathcal{P}_1) & \equiv & \{\ \mathit{dcl}(\mathtt{A::f}),\ \mathit{dcl}(\mathtt{A::g}),\ \mathit{dcl}(\mathtt{A::x}),\ \mathit{dcl}(\mathtt{B::g}), \\ & & \ \ \mathit{dcl}(\mathtt{B::y}),\ \mathit{dcl}(\mathtt{C::f}),\ \mathit{dcl}(\mathtt{C::z})\ \} \\ \mathit{MemberDefs}(\mathcal{P}_1) & \equiv & \{\ \mathit{def}(\mathtt{A::f}),\ \mathit{def}(\mathtt{A::g}),\ \mathit{def}(\mathtt{B::g}),\ \mathit{def}(\mathtt{C::f})\ \} \end{array}$$

In Section 6.2, we will discuss how class-typed data members (which behave like variables because other members can be accessed from them) are modeled.

4. TABLE CONSTRUCTION

This section describes how tables and lattices are constructed. Recall that the purpose of the table is to record for each variable the set of members that are used. A few auxiliary definitions will be presented first, in Section 4.1.

4.1 Auxiliary Definitions

For each variable v in *ClassPtrVars*($\mathcal{P}$) we will need a conservative approximation of the variables in *ClassVars*($\mathcal{P}$) variables that v may point to. Any of several existing algorithms [Andersen 1994; Choi et al. 1993; Pande and Ryder 1996; Steensgaard 1996; Shapiro and Horwitz 1997; Das 2000] can be used to compute this information, and we do not make assumptions about the particular algorithm used to compute points-to information. Definition 3 expresses the information supplied by some points-to analysis algorithm as a set $\mathit{PointsTo}(\mathcal{P})$, which contains a pair $\langle p, v\rangle$ for each pointer p that may point to a class-typed variable v.

Definition 3. Let $\mathcal{P}$ be a program. Then, the points-to information for $\mathcal{P}$ is defined as follows:

$$\mathit{PointsTo}(\mathcal{P}) \triangleq \{ \langle p, v\rangle \mid p \in \mathit{ClassPtrVars}(\mathcal{P}), v \in \mathit{ClassVars}(\mathcal{P}), p \text{ may point to } v \}$$

Example. We will use the following points-to information for program $\mathcal{P}_1$. Recall that $X{::}f$ denotes the `this` pointer of method $X{::}f()$.

$$\begin{array}{l} \mathit{PointsTo}(\mathcal{P}_1) \equiv \\ \quad \{ \langle \mathtt{ap}, \mathtt{a}\rangle, \langle \mathtt{ap}, \mathtt{b}\rangle, \langle \mathtt{ap}, \mathtt{c}\rangle, \langle \mathtt{A::f}, \mathtt{a}\rangle, \langle \mathtt{A::f}, \mathtt{b}\rangle, \langle \mathtt{C::f}, \mathtt{c}\rangle, \langle \mathtt{A::g}, \mathtt{a}\rangle, \langle \mathtt{B::g}, \mathtt{b}\rangle, \langle \mathtt{B::g}, \mathtt{c}\rangle \} \end{array}$$

Note that the following simple algorithm suffices to compute the information for this example: for each pointer p of type $*X$, assume that it may point to any object of type Y, such that (i) $Y = X$ or Y is a class transitively derived from X, and (ii) if p is the `this` pointer of a virtual method $C{::}m$, no overriding definitions of m are visible in class Y.

We will use the following terminology for function and method calls. A *direct* call is any call to a function or a nonvirtual method, or an invocation of a virtual method from a variable in $\mathit{ClassVars}(\mathcal{P})$. An *indirect* call is an invocation of a virtual method from a variable in $\mathit{ClassPtrVars}(\mathcal{P})$ (requiring a dynamic dispatch).

4.2 Table Entries for Member Access Operations

Table T has a *row* for each element of $\mathit{ClassVars}(\mathcal{P})$ and $\mathit{ClassPtrVars}(\mathcal{P})$, and a *column* for each element of $\mathit{MemberDcls}(\mathcal{P})$ and $\mathit{MemberDefs}(\mathcal{P})$. Informally, an entry $(y, \mathit{dcl}(A{::}m))$ appears in T iff the declaration of m is contained in y's type, and an entry $(y, \mathit{def}(A{::}m))$ appears in T iff the definition of m is contained in y's type. We begin by adding entries to T that reflect the member access operations in the program. Definition 4 below defines a set $\mathit{MemberAccess}(\mathcal{P})$ of all pairs $\langle m, y\rangle$ such that member m is accessed from variable y. For an *indirect* call $p \rightarrow f(y_1, \ldots, y_n)$, we also include an element $\langle f, y\rangle$ in $\mathit{MemberAccess}(\mathcal{P})$ for each $\langle p, y\rangle \in \mathit{PointsTo}(\mathcal{P})$.

Definition 4. Let $\mathcal{P}$ be a program. Then, the set of member access operations in $\mathcal{P}$ is defined as follows:

$$\begin{array}{l} \mathit{MemberAccess}(\mathcal{P}) \triangleq \\ \quad \{ \langle m, v\rangle \mid v.m \text{ occurs in } \mathcal{P},\ m \text{ is a class member in } \mathcal{P}, v \in \mathit{ClassVars}(\mathcal{P}) \} \cup \\ \quad \{ \langle m, *p\rangle \mid\ p \rightarrow m \text{ occurs in } \mathcal{P},\ m \text{ is a class member in } \mathcal{P}, \\ \qquad\qquad p \in \mathit{ClassPtrVars}(\mathcal{P}) \} \cup \\ \quad \{ \langle m, y\rangle \mid p \rightarrow m \text{ occurs in } \mathcal{P},\ \langle p, y\rangle \in \mathit{PointsTo}(\mathcal{P}), m \text{ is a virtual method in } \mathcal{P} \} \end{array}$$

Example. For program $\mathcal{P}_1$ of Figure 4, we have

$$\begin{array}{l} \mathit{MemberAccess}(\mathcal{P}_1) \equiv \\ \quad \{ \langle \mathtt{x}, \mathtt{*A::g}\rangle, \langle \mathtt{y}, \mathtt{*B::g}\rangle, \langle \mathtt{z}, \mathtt{*C::f}\rangle, \langle \mathtt{g}, \mathtt{*A::f}\rangle, \langle \mathtt{g}, \mathtt{*C::f}\rangle, \\ \quad\ \ \langle \mathtt{f}, \mathtt{*ap}\rangle, \langle \mathtt{f}, \mathtt{a}\rangle, \langle \mathtt{f}, \mathtt{b}\rangle, \langle \mathtt{f}, \mathtt{c}\rangle, \langle \mathtt{g}, \mathtt{a}\rangle, \langle \mathtt{g}, \mathtt{b}\rangle, \langle \mathtt{g}, \mathtt{c}\rangle \} \end{array}$$

Accessing a class member is not an entirely trivial operation because different classes in a class hierarchy may contain members with the same name (or signature). Furthermore, in the presence of multiple inheritance, an object may contain multiple subobjects of a given type C, and hence multiple members C::m. This implies that whenever a member m is accessed, one needs to determine which m is being selected. This selection process is defined informally in the C++ Draft Standard [Accredited Standards Committee X3 1997] as a set of rules that determine when a member *hides* or *dominates* another member with the same name. Rossie and Friedman [1995] provided a formalization of the member lookup, as a function on *subobject graphs*. This framework has subsequently been used by Tip et al. as a formal basis for operations on class hierarchies such as slicing [Tip et al. 1996] and specialization [Tip and Sweeney 1997; 2000].

For the purposes of the present article, we will assume the availability of a function *static-lookup* which, given a class C and a member m, determines the base class B (B is either C, or a transitive base class of C) in which the selected member is located.[6] For details on function *static-lookup*, the reader is referred to Rossie and Friedman [1995] and Tip et al. [1996].

We are now in a position to state how the appropriate relations between variables and declarations and definitions should be added to the table:

Definition 5. Let $\mathcal{P}$ be a program with associated table T. Then, the following entries are added to the table due to member access operations that occur in the program.

$$\frac{\begin{array}{c}\langle m, y\rangle \in \mathit{MemberAccess}(\mathcal{P}),\ m \in \mathit{DataMembers}(\mathcal{P}),\\ X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(\mathcal{P}, y), m)\end{array}}{(y, \mathit{dcl}(X{::}m)) \in T}$$

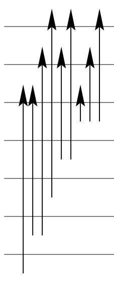
$$\frac{\begin{array}{c}\langle m, y\rangle \in \mathit{MemberAccess}(\mathcal{P}),\ m \in \mathit{NonVirtualMethods}(\mathcal{P}),\\ X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(\mathcal{P}, y), m)\end{array}}{(y, \mathit{def}(X{::}m)) \in T}$$

$$\frac{\begin{array}{c}\langle m, y\rangle \in \mathit{MemberAccess}(\mathcal{P}),\ m \in \mathit{VirtualMethods}(\mathcal{P}),\\ y \equiv *p,\ p \in \mathit{ClassPtrVars}(\mathcal{P}),\\ X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(\mathcal{P}, y), m)\end{array}}{(y, \mathit{dcl}(X{::}m)) \in T}$$

$$\frac{\begin{array}{c}\langle m, y\rangle \in \mathit{MemberAccess}(\mathcal{P}),\ m \in \mathit{VirtualMethods}(\mathcal{P}),\\ y \equiv v,\ v \in \mathit{ClassVars}(\mathcal{P}),\\ X \equiv \mathit{static\text{-}lookup}(\mathit{TypeOf}(\mathcal{P}, y), m)\end{array}}{(y, \mathit{def}(X{::}m)) \in T}$$

[6] In Rossie and Friedman [1995] and Tip et al. [1996], *static-lookup* is defined as a function from subobject to subobjects. Since the present article is only concerned with the *classes* in which members are located, we will simply ignore all subobject information below.

Table I. Initial Table for Program $\mathcal{P}_1$ of Figure 4. Arrows indicate implications due to assignments (see Section 4.4).



	dcl(A::f)	dcl(A::g)	dcl(A::x)	def(A::f)	def(A::g)	dcl(B::g)	dcl(B::y)	def(B::g)	dcl(C::z)	def(C::f)
a				×	×					
b				×				×		
c								×		×
*ap	×									
*A::f		×		×						
*A::g			×		×					
*B::g							×	×		
*C::f						×			×	×

4.3 Table Entries for `this` Pointers

The next table construction rule we will present is concerned with `this` pointers of methods. Consider the fact that for each method C::f(), there is a column in the table labeled *def*(C::f), and a row labeled $*C$::f. The former is used to express the fact that method C::f() may be called from objects. The latter is necessary to reflect members being accessed from method C::f()'s `this` pointer. Unless precautions are taken, the attribute *def*(C::f) and the object $*C$::f may appear at different points in the lattice, though $\gamma(*C::f) \geq \mu(\mathit{def}(C::f))$ must always hold.[7] In such cases, our method effectively infers that the type of a `this` pointer could be a base class of the type in which method C::f occurs (and therefore be less constrained). However, in reality, the type of a method's `this` pointer is *determined by* the class in which the associated method definition appears.

The table entries added by Definition 6 will force a method's attribute and a method's `this` pointer to appear at the same lattice element, by ensuring $\gamma(*C::f) \leq \mu(\mathit{def}(C::f))$. This will allow us later to remove rows for `this` pointers from the table when constructing the lattice.

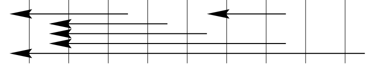
Definition 6. Let $\mathcal{P}$ be a program. Then, the following entries are added to the table:

$$\frac{\mathit{def}(C::m) \in \mathit{MemberDefs}(\mathcal{P})}{(*C::m, \mathit{def}(C::m)) \in T}$$

Example. Table I shows the table for program $\mathcal{P}_1$ of Figure 4 after adding the entries according to Definitions 5 and 6. The purpose of the arrows at the side of the table will be explained in Section 4.4.

Table II. Table after application of Assignment Implications. Arrows indicate implications for preserving hiding/dominance among members with the same name (see Section 4.5).

	`dcl(A::f)`	`dcl(A::g)`	`dcl(A::x)`	`def(A::f)`	`def(A::g)`	`dcl(B::g)`	`dcl(B::y)`	`def(B::g)`	`dcl(C::z)`	`def(C::f)`
`a`	×	×	×	×	×					
`b`	×	×		×			×	×		
`c`	×					×	×	×	×	×
`*ap`	×									
`*A::f`		×		×						
`*A::g`			×		×					
`*B::g`							×	×		
`*C::f`						×			×	×



4.4 Table Entries for Assignments

Consider an assignment $x = y$, where $x \equiv v$ and $y \equiv w$, for some class-typed variables v, $w \in \mathit{ClassVars}(\mathcal{P})$. Such an assignment is only valid if the type of x is a base class of the type of y. Consequently, any member declaration or definition that occurs in x's type must also occur in y's type. We will enforce this constraint using an *implication* from the row for x to the row for y. However, we will begin by formalizing the notion of an assignment.

Definition 7 below defines a set $\mathit{Assignments}(\mathcal{P})$ that contains a pair of objects $\langle v, w\rangle$ for each assignment $v = w$ in $\mathcal{P}$ where v and w are class-typed. In addition, $\mathit{Assignments}(\mathcal{P})$ also contains entries for cases where the type of the left-hand side and/or the right-hand side of the assignment are a pointer to a class. Parameter-passing in direct calls to functions and methods is modeled by way of assignments between corresponding formal and actual parameters. For an *indirect* call $p \rightarrow f(y_1, \ldots, y_n)$, $\mathit{Assignments}(\mathcal{P})$ contains additional elements that model the parameter-passing in the direct call $x.f(y_1, \ldots, y_n)$, for each $\langle p, x\rangle \in \mathit{PointsTo}(\mathcal{P})$. That is, we conservatively approximate the potential targets of dynamically dispatched calls. The set $\mathit{Assignments}(\mathcal{P})$ will also contain elements for implicit parameters such as `this` pointers of methods and function/method return values whose type is class-related.

Definition 7. Let $\mathcal{P}$ be a program. Then, the set of assignments between vari-

[7]See Appendix.

```
class A { m(); ...};
class B : A { m(); ...};
A a; B b; B c;
a.m(); b.m(); c.m();
a = b;
```

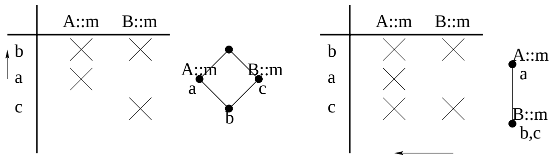


Fig. 5. Effect of dominance rules.

ables whose type is a (pointer to a) class is defined as follows:

$$
\begin{aligned}
&\mathit{Assignments}(\mathcal{P}) \triangleq \\
&\quad \{\, \langle v, w\rangle \mid v = w \text{ occurs in } \mathcal{P},\ v,\ w \in \mathit{ClassVars}(\mathcal{P}) \,\} \cup \\
&\quad \{\, \langle *p, w\rangle \mid\ p = \&w \text{ occurs in } \mathcal{P},\ p \in \mathit{ClassPtrVars}(\mathcal{P}), w \in \mathit{ClassVars}(\mathcal{P}) \,\} \cup \\
&\quad \{\, \langle *p, *q\rangle \mid p = q \text{ occurs in } \mathcal{P},\ p,\ q \in \mathit{ClassPtrVars}(\mathcal{P}) \,\} \cup \\
&\quad \{\, \langle *p, w\rangle \mid\ *p = w \text{ occurs in } \mathcal{P},\ p \in \mathit{ClassPtrVars}(\mathcal{P}), w \in \mathit{ClassVars}(\mathcal{P}) \,\} \cup \\
&\quad \{\, \langle v, *q\rangle \mid\ v = *q \text{ occurs in } \mathcal{P},\ v \in \mathit{ClassVars}(\mathcal{P}), q \in \mathit{ClassPtrVars}(\mathcal{P}) \,\} \cup \\
&\quad \{\, \langle *p, *q\rangle \mid *p = *q \text{ occurs in } \mathcal{P},\ p,\ q \in \mathit{ClassPtrVars}(\mathcal{P}) \,\}
\end{aligned}
$$

Example. For program $\mathcal{P}_1$ of Figure 4, we have

$$
\begin{aligned}
&\mathit{Assignments}(\mathcal{P}_1) \equiv \\
&\quad \{\ \langle \texttt{*ap}, \texttt{a}\rangle,\ \langle \texttt{*ap}, \texttt{b}\rangle,\ \langle \texttt{*ap}, \texttt{c}\rangle,\ \langle \texttt{*A::f}, \texttt{a}\rangle,\ \langle \texttt{*A::f}, \texttt{b}\rangle, \\
&\qquad \langle \texttt{*C::f}, \texttt{c}\rangle,\ \langle \texttt{*A::g}, \texttt{a}\rangle,\ \langle \texttt{*B::g}, \texttt{b}\rangle,\ \langle \texttt{*B::g}, \texttt{c}\rangle\ \}
\end{aligned}
$$

We are now in a position to express how elements should be added to the table due to assignments. Definition 8 states this as an *implication*, which tells us how elements should be copied from one row to another.

Definition 8. Let $\mathcal{P}$ be a program with associated table T. Then, the following implications must be encoded in the table due to assignments that occur in $\mathcal{P}$:

$$\frac{\langle x, y\rangle \in \mathit{Assignments}(\mathcal{P})}{x \rightarrow y}$$

Note that assignment implications are implications between "objects" (in the sense of concept analysis); hence an assignment implication $x \rightarrow y$ causes x to appear above y (i.e., $\gamma(x) \geq \gamma(y)$) in the lattice. Cyclic assignments will generate cyclic implications, which will collapse the corresponding lattice elements into one point: all the involved variables must have the same type.

Example. For program $\mathcal{P}_1$ of Figure 4, the following assignment implications are generated:

$$
\begin{aligned}
&\texttt{*ap} \rightarrow \texttt{a},\ \texttt{*ap} \rightarrow \texttt{b},\ \texttt{*ap} \rightarrow \texttt{c},\ \texttt{*A::f} \rightarrow \texttt{a},\ \texttt{*A::f} \rightarrow \texttt{b}, \\
&\texttt{*C::f} \rightarrow \texttt{c},\ \texttt{*A::g} \rightarrow \texttt{a},\ \texttt{*B::g} \rightarrow \texttt{b},\ \texttt{*B::g} \rightarrow \texttt{c}
\end{aligned}
$$

These implications are indicated on the left side of Table I. Table II is obtained by copying the elements from the "source row" to the "target row" according to each of these implications.

4.5 Table Entries for Preserving Dominance/Hiding

The table thus far encodes for each variable the members contained in its type (either directly because a member is accessed from that variable, or indirectly due to assignments between variables). However, in the original class hierarchy, an object's type may contain more than one member with a given name. In such cases, the member lookup rules of [Accredited Standards Committee X3 1997] determine which member is accessed. This is expressed as a set of rules that determine when a member *hides* or *dominates* another member with the same name. In cases where a variable contains two members m that have a hiding relationship in the original class hierarchy, this hiding relationship must be preserved: we are interested in generating a restructured hierarchy from the table, and the member access operations in the program might otherwise become ambiguous. Definition 9 incorporates the appropriate hiding/dominance relations into the table, using implications between attributes:

Definition 9. Let $\mathcal{P}$ be a program with associated table T. Then, the following implications are incorporated into T in order to preserve hiding and dominance:

$$\frac{(x, \mathit{dcl}(A{::}m)) \in T,\ (x, \mathit{dcl}(B{::}m)) \in T,\\ A \text{ is a transitive base class of } B}{\mathit{dcl}(B{::}m) \rightarrow \mathit{dcl}(A{::}m)}$$

$$\frac{(x, \mathit{dcl}(A{::}m)) \in T,\ (x, \mathit{def}(B{::}m)) \in T,\\ A = B \text{ or } A \text{ is a transitive base class of } B}{\mathit{def}(B{::}m) \rightarrow \mathit{dcl}(A{::}m)}$$

$$\frac{(x, \mathit{def}(A{::}m)) \in T,\ (x, \mathit{def}(B{::}m)) \in T,\\ A \text{ is a transitive base class of } B}{\mathit{def}(B{::}m) \rightarrow \mathit{def}(A{::}m)}$$

$$\frac{(x, \mathit{def}(A{::}m)) \in T,\ (x, \mathit{dcl}(B{::}m)) \in T,\\ A \text{ is a transitive base class of } B}{\mathit{dcl}(B{::}m) \rightarrow \mathit{def}(A{::}m)}$$

Dominance implications are implications between "attributes" (in the sense of concept analysis); hence a dominance implication $B{::}m \rightarrow A{::}m$ will cause $B{::}m$ to appear below $A{::}m$ (i.e., $\mu(B{::}m) \leq \mu(A{::}m)$) in the lattice. Due to the condition "A is a (transitive) base class of B," dominance implications always connect subclass members to superclass members and cannot contain cycles (if $A = B$, a one-point "cycle" is generated). Note the symmetry between assignment implications and dominance implications: the former are implications between rows (objects) and serve to preserve behavior of subobject selection; the latter are implications between columns (attributes) and serve to preserve behavior of member lookup. Figure 5 demonstrates the effect of the dominance rules: subclass B of class A redefines method m. In the table, the implication due to assignment $a = b$; forces row a to be added to row b. But now the member access $b.m()$ has become ambiguous: the row for b contains entries for both $A{::}m$ and $B{::}m$. According to the dominance rules, an implication $B{::}m \rightarrow A{::}m$ is generated, which adds entry $(b, A{::}m)$ to the

	dcl(A::f)	dcl(A::g)	dcl(A::x)	def(A::f)	def(A::g)	dcl(B::g)	dcl(B::y)	def(B::g)	dcl(C::z)	def(C::f)
a	×	×	×	×	×					
b	×	×		×		×	×	×		
c	×	×				×	×	×	×	×
*ap	×									
*A::f	×	×		×						
*A::g		×	×		×					
*B::g		×				×	×	×		
*C::f	×	×				×			×	×

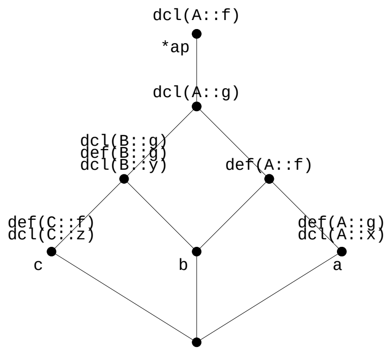


Fig. 6. Final table and lattice for program $\mathcal{P}_1$, after removing the rows labeled `*A::f`, `*A::g`, `*B::g`, and `*C::f`..

table. The corresponding lattice is a two-element chain and thus reproduces the original hierarchy, thereby reestablishing the dominance of $B::m$ over $A::m$.

Example. For program $\mathcal{P}_1$, the following dominance implications are generated:

$$\begin{array}{lll} \mathit{def}(\texttt{A::f}) \rightarrow \mathit{dcl}(\texttt{A::f}) & \mathit{def}(\texttt{A::g}) \rightarrow \mathit{dcl}(\texttt{A::g}) & \mathit{def}(\texttt{B::g}) \rightarrow \mathit{dcl}(\texttt{A::g}) \\ \mathit{dcl}(\texttt{B::g}) \rightarrow \mathit{dcl}(\texttt{A::g}) & \mathit{def}(\texttt{B::g}) \rightarrow \mathit{dcl}(\texttt{B::g}) & \mathit{def}(\texttt{C::f}) \rightarrow \mathit{dcl}(\texttt{A::f}) \end{array}$$

These implications are shown at the bottom of Table II. After incorporating these implications, the table in Figure 6 results.

Remark. Observe that the implication *def*(`B::g`) → *dcl*(`A::g`) only becomes necessary after propagating table elements according to the other implications.

5. THE NEW HIERARCHY

5.1 Lattice Construction

Since the assignment implications can generate new dominance implications and vice versa, a fixpoint iteration is necessary in order to compute the final table. This algorithm is described in Section 8. After the table has converged, the lattice is constructed using Ganter's algorithm (see Section 2). As explained above, it can be interpreted directly as a new class hierarchy.

There is one issue concerning pointers that deserves mentioning. Recall that in Section 4.3 table entries were added to ensure that method definitions and their `this` pointers show up at the same lattice element. In order to avoid presenting redundant information to the user, we will henceforth omit `this` pointers from the lattice. The easiest way to accomplish this is to remove the rows for `this` pointer variables to the table prior to generating the lattice. Note that rows for `this` pointers cannot be left out during table construction because they are needed to model member accesses from `this` pointers, and the elements in such rows may be involved in implications due to assignments and dominance relations.

Example. Figure 6 shows the lattice for program $\mathcal{P}_1$, generated from the final table after removing the rows labeled `*A::f`, `*A::g`, `*B::g`, and `*C::f`. The lattice can be interpreted directly as a new class hierarchy. It demonstrates that `a` does not access `B::y` and `C::z`, while `b` and `c` do not access `A::x` and `b` does not access

`C::z`. Similarly, the lattice shows the fine-grained differences in method access: for example, `c` does not need `def(A::f)` and `def(A::g)`. Thus `a`, `b`, `c` will receive new types. The program statements are unchanged, but according to the new hierarchy, both `b` and `c` have become smaller.

Note that from a space optimization viewpoint, the lattice can be simplified further: for example, the two topmost elements could be merged (as they only contain method declarations), and even the edge $b \leftrightarrow def(A{::}f)$ could be merged with the parallel edge. Tip and Sweeney [2000] discuss such "peephole optimizations" in detail.

5.2 Properties of the Lattice

The lattice, being a concept lattice, enjoys several important properties [Ganter and Wille 1999]:

—The lattice is the smallest lattice compatible with the table and thus can be order-embedded into any other lattice compatible with the table. In fact, if the table represents a partial order, then the lattice is the Dedekind-McNeill completion of this partial order. Hence, the lattice is *minimal*.

—Attribute labels of lattice elements always occur as far upward in the lattice as possible (see Section 2). Since attribute labels correspond to members in the classes of the new hierarchy, common members are factored out as much as possible. The same applies to common variables, which are factored out downward as much as possible. Therefore the lattice is *maximally factorized*.

More important than minimality and maximal factorization are semantic properties of the lattice. In Tip and Sweeney [2000] it was proved that the assignment constraints and the dominance constraints guarantee

—*preservation of assignment behavior*: every assignment will select the same subobject from the right-hand-side object as in the original program;

—*preservation of lookup behavior*: every method call will select the same method definition via dynamic lookup as in the original program.

The lattice, interpreted as a new class hierarchy, respects all assignment and dominance constraints by construction. Furthermore, by construction, in the new hierarchy a member is visible to an object if and only if the object accesses the member. Since the statements of the program are unchanged, we thus can guarantee that *the new hierarchy is operationally equivalent to the old one.* This fact is explained in more detail in Tip and Sweeney [2000].

The lattice may contain elements which are neither labeled with an attribute nor an object (e.g., the center element in Figure 3). Such elements are called "empty" and serve merely to group the members of other classes. In our application, an empty element C corresponds to a class which neither has any members, nor does any variable have type C in the new hierarchy. Section 7 will explain how to eliminate empty elements.

Remember that rows for `this` pointers have been removed from the final table without semantic effect. A similar simplification can be applied to pointers in general. The lattice may be very fine-grained due to access patterns of pointers which basically have the same type, but access different members. Since a pointer will always appear above any object it may point to (see proof in the Appendix), rows for pointers can safely be removed from the final table. The pointers can then be given the same type as the objects they point to—which still guarantees operational equivalence, since any pointer may still access all members it needs.[8] Of course, pointer rows are essential during table generation, as explained above for the special case of `this` pointers.

How does the final lattice depend on the precision of the points-to analysis? Since any points-to analysis computes a conservative approximation, different analyses differ only with respect to the row entries they generate for pointer variables: the more precise the points-to analysis, the less entries any pointer row will have. That is, if table T_1 has been generated using a more precise points-to analysis than for table T_2, we have $(o, a) \in T_1 \Rightarrow (o, a) \in T_2$. By the fundamental property $(o, a) \in T_{1,2} \iff \gamma_{1,2}(o) \leq \mu_{1,2}(a)$, hence $\gamma_1(o) \leq \mu_1(a) \Rightarrow \gamma_2(o) \leq \mu_2(a)$, thus $\mathcal{L}(T_1)$ can be order-embedded into $\mathcal{L}(T_2)$. In $T_{1,2}$, any possible pointer row is a superset of a "minimal row" for that pointer, which corresponds to the (undecidable) limit case of precise points-to analysis. By the above embedding, the lattice for the limit case can be found as a substructure in every actual lattice.

6. LANGUAGE DETAILS

The basic process for constructing tables and lattices, as described in the previous section, did not address a number of language features, that are not core issues, but that are indispensable in practice. This section addresses a number of such issues.

6.1 Heap Allocation

We handle heap-allocated objects in a straightforward way by simply treating each allocation site in the program as a class-typed variable (e.g., an element of the set *ClassVars*). For the program of Figure 1, there are four such allocation sites, which we refer to as `Student1`, `Student2`, `Professor1`, and `Professor2`. In principle, more sophisticated context-sensitive analyses could be used to distinguish heap-allocated objects in different calling contexts, but we expect the benefits of this additional precision to be limited.

6.2 Modeling Nested Objects

The treatment of *class-related data members* (i.e., data members whose type is class-related such as `Student::advisor` in Figure 1) is an important issue. Like data members of built-in types, class-related data members can be accessed from variables and are therefore modeled as attributes. However, since other members may be accessed from a class-related data member, such data members play an alternate role as objects.

In order to clarify the issues involved in the reengineering of such "nested" structures, consider a class C that contains a data member m whose type is some class D. Then, the following information about m is made explicit in the concept lattice:

—The set of variables in which m is contained. This is modeled by treating m as an "attribute" a. Any object that occurs below a in the lattice contains m.

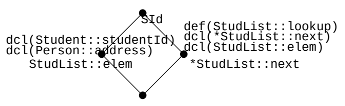
[8]More precisely, the pointer is given the supremum type of all object types it may point to; this supremum also exists in the reduced lattice and is still below the pointer's type in the full lattice.

```
class StudList {
 public:
  Student elem;
  StudList *next;
  Address lookup(long SId){
    if (elem.studentId==SId)
      return elem.address;
    else
      return
        next->lookup(SId);
  }
}
```

(a)

	def(StudList::lookup)	dcl(*StudList::next)	dcl(StudList::elem)	dcl(Student::studentId)	dcl(Person::address)
StudList::elem				×	×
*StudList::next	×	×	×		
SId					
*StudList::lookup	×	×	×		

(b)



(c)

Fig. 7. Analyzing a linked list of students.

—The set of members contained in the type of m. This is modeled by treating the type of m as an "object" o. The set of members contained in o corresponds to the attributes that occur above o in the lattice. This set of members is a subset of the members of D in the original class hierarchy.

Note that the "attribute view" of m corresponds exactly to the way we previously modeled data members with a built-in type, whereas the "object view" of m corresponds exactly to the way we previously modeled variables. The definitions that are concerned with variables therefore apply to class-related data members as well, and for convenience we will henceforth assume the term "variable" to include class-related data members.

Figure 7(a) shows an example that illustrates the issues related to class-related data members. Here, the program of Figure 1 is extended with a linked list of students. Observe that the data members `address` and `studentId` are accessed from the `elem` member of a student list, and that the `lookup` method of class `StudList` is accessed from the `next` data member. The data members `elem` and `next` are accessed from method `lookup`'s `this` pointer, and as usual, a table entry is added for the associated method definition of `lookup`. Furthermore, there is an assignment implication `*StudList::lookup = next` due to the recursive call.

Figure 7(b) shows the table after applying this implication, and Figure 7(c) shows the associated lattice. Let us call the left element *Student′* and the right element *StudList′*, as suggested by their labels. Both `next` resp. `elem` show up twice in the lattice. *Student′* shows that the *type* of `StudList::elem` must be *Student′*; *StudList′* shows that the *member* `StudList::elem` must be located within class *StudList′*. Similar observations are valid for `next`: it must be located in *StudList′* and has type *StudList′*—as to be expected. The example thus illustrates the dual role of class-typed data members: the lattice will not only display their type, but also their position in the class hierarchy.

6.3 Modeling Constructors

Constructors require special attention. A constructor generally initializes all data members contained in an object. If no constructor is provided by the user, a

```
class O {
  ...
};

class A extends O {
  ...
};

class B extends O {
  ...
};

class Example {
  public static void main(String args[]){
    A a = new A();
    O o = a;

    if (o instanceof A){
      /* reached */
    }

    if (o instanceof B){
      /* unreached */
    }

    A a2 = (A)o;  /* succeeds */
    B b = (B)o;  /* ClassCastException */
  }
}
```

(a)

```
class O {
  boolean isA(){ return false; };
  boolean isB(){ return false; };
  A toA(){ throw new ClassCastException(); };
  B toB(){ throw new ClassCastException(); };
};

class A extends O {
  boolean isA(){ return true; };
  A toA(){ return this; };
  ...
};

class B extends O {
  boolean isB(){ return true; };
  B toB(){ return this; };
  ...
};

class Example {
  public static void main(String args[]){
    A a = new A();
    O o = a;

    if (o.isA()){
      /* reached */
    }

    if (o.isB()){
      /* unreached */
    }

    A a2 = o.toA();  /* succeeds */
    B b  = o.toB();  /* ClassCastException */
  }
}
```

(b)

Fig. 8. (a) Example Java program that uses type cast and `instanceof` operations. (b) Equivalent Java program after transforming away `instanceof` and cast operations.

so-called default constructor is generated by the compiler, which performs the necessary initializations. The compiler may also generate a *call* to a constructor in certain cases. Modeling these compiler-generated actions as member access operations would lead us to believe that each member m of class C is needed in all C-instances, even in cases where the only access to m consists of its (default) initialization. Compiler-generated constructors, compiler-generated initializations, and compiler-generated calls to constructors will therefore be excluded from the set of member access operations. Destructors can be handled similarly.

6.4 Type Casts and Type Test Operations

Modern object-oriented languages such as Java provide language features for testing the run-time type of an object, or down-casting an object to a derived type. Since these operations are used heavily, any realistic implementation will need to deal with them. Note that a “catch” statement also may perform implicit type test operations.

Our approach to dealing with type cast and instance-of operations will be to transform them into a semantically equivalent piece of code consisting of only virtual method calls and exception-handling constructs. We will outline these transformations for the cast and instanceof operations as they are used in Java. In Java, these operations have the following semantics:

—An expression e instanceof C evaluates to true if the run-time type of the object pointed to by reference e is C or a subclass of C. Otherwise, the expression evaluates to false.

—A cast-expression (C)e evaluates to an expression e with static type C if the run-time type of e is C or a subclass of C. Otherwise, an exception of type ClassCastException is thrown.

Our strategy for transforming instanceof-expressions will be as follows.[9] For each type C, we introduce a method isC() in the root class O of the hierarchy. This method has return type boolean, and the default definition of isC() in class O returns false. Class C provides an overriding definition[10] of isC() that returns true. Now, every expression of the form e instanceof C is transformed into e.isC(). One can see easily that the expressions e instanceof C and e.isC() return true under exactly the same conditions.

Cast expressions are transformed in a similar manner. For each type C, we introduce a method toC() in the root class O of the hierarchy. This method has return type C, and the default definition of isC in class O throws an exception of type ClassCastException. Class C provides an overriding definition of toC() that returns this. Now, every expression of the form (C)e is transformed into e.toC(). It can easily be seen that the expressions (C)e and e.toC() succeed under exactly the same conditions.

Figure 8(a) shows an example program containing various instanceof and downcast expressions. Figure 8(b) shows the program after transforming away all these expressions. After eliminating all cast and instanceof expressions, the resulting program can be processed with the techniques presented in the previous section.

After generating the lattice, the artificial is$C$() and to$C$() methods can easily be transformed back into cast and instanceof operations if the reengineer desires to do so. As an example, we will outline how to$C$() methods can be transformed back into cast operations in the transformed class hierarchy. Let x be the lattice element labeled def(C.to$C$()), and suppose that class name X has been associated with this lattice element. Then e.to$C$() can be transformed into (X)e.

6.5 Exceptions

Exception-handling constructs give rise to additional control flow, additional assignments between the thrown object and the parameter in a matching catch clause, and additional instanceof expressions[11], but do not influence member access patterns. Therefore, the previously discussed mechanisms suffice. Note, however, that the abundance of (implicit or explicit) exceptions in Java complicates points-to analysis, and one might think of adopting factored control flow graphs [Choi et al. 1999] for our analysis.

[9]This transformation was proposed by M. Streckenbach.
[10]In cases where the target type C is an interface, this overriding definition should not be placed in C itself, but in all classes that implement C.
[11]A catch clause catch (E e){ $\cdots$ } implicitly performs a run-time type test e instanceof E.

6.6 Arrays

Arrays are treated as monolithic variables, and we do not distinguish between different array elements. One might think of integrating a fine-grained array analysis, such as the Omega test [Pugh and Wonnacot 1998].

However, fine-grained array analysis may also reduce analysis precision in case of mixed co- and contravariance. In Java, arrays are covariant: $A \leq B$ implies $A[] \leq B[]$. If Java would allow fully contravariant method overriding (which it does not), we would have $dom(f_A) \geq dom(f_B)$ for any B-method f redefined in A. Our approach is to translate array accesses into predefined method calls $a.store(i, x)$ and $a.access(i)$.[12] Thus, by contravariance for *store*'s second argument we would obtain $A = dom(store_{A[]})[2] \geq dom(store_{B[]})[2] = B$. Hence, we suddenly have $A \leq B$ as well as $B \leq A$, collapsing lattice elements and hiding fine-grained access patterns.

6.7 Dynamic Class Loading

Java offers a mechanism for dynamic loading of classes, as well as some reflective devices. For example, it is possible to construct a string at run-time, interpret that string as the name of a class, and create an object of that type. Similarly, methods can be invoked by supplying run-time values that represent their name and signature. Since reflection and dynamic loading access a program construct via its name, it limits our capabilities for manipulating the class hierarchy—any component that is accessed via reflection has to be preserved. In general, it is impossible to determine the program constructs that are accessed by way of reflection and dynamic loading because run-time values are used. Therefore, it is clear that in the absence of additional information, worst-case assumptions have to be made that would result in a massive loss of analysis precision.

However, in many cases dynamic loading is not used in an arbitrary way. For example, the jEdit program (see Section 9.3) uses dynamic class loading just for configuration management: in order to tailor the program to a specific context, some classes from a statically determined set of classes are loaded dynamically. In such cases, the complete set can be analyzed together with the rest of the program, retaining analysis precision.

A general solution would be to rely on additional input from the user that specifies where reflection and dynamic loading are used in an application. Sweeney and Tip [2000] present a small specification language for providing information about reflective features in the context of an application extraction tool [Tip et al. 1999], which could be adapted for our purposes in principle. Of course, in cases where incorrect or incomplete information is provided, operational equivalence is lost. However, it is possible to insert run-time checks in the generated code to catch such problems quickly.

6.8 Multiple Subobjects

If an object x contains multiple subobjects of some type C (due to the use of nonvirtual multiple inheritance), our tables do not make a distinction between the

[12]This transformation is similar to the transformations for typecasts and `instanceof` expressions described above.

Table III. Final Table for the Student/Professor Example

	dcl(Person::name)	dcl(Person:address)	dcl(Person::socialsecurityNumber)	dcl(Student::studentId)	dcl(Student::advisor)	dcl(Professor::faculty)	dcl(Professor::workAdress)	dcl(Profesor::assistant)	def(Student::Student)	def(Student::setAdvisor)	def(Professor::Professor)	def(Professor::hireAssistant)
*s1					X					X		
*s2												
*p1												
*p2								X				X
*s												
*p												
Student1	X	X		X	X				X	X		
Student2	X	X		X					X			
Professor1	X					X	X	X			X	
Professor2	X					X	X	X			X	X
*advisor												
*assistant												
*Student::Student	X	X		X					X			
*Student::setAdvisor					X					X		
*Professor::Professor	X					X	X	X			X	
*Professor::hireAssistant								X				X

various "copies" of the members of C in x. This leads to problems if the objective is to generate a new hierarchy from the lattice in which the distinct copies of the members of C must be preserved. We consider this to be a minor problem because situations where nonvirtual inheritance is used for its "member replicating" effect are quite rare in practice, and the restructuring tool could inform the user of the cases where the problem occurs. A clean solution to this problem would involve the encoding of subobject information in the table using an adaptation of the approach of Tip and Sweeney [1997; 2000].

7. RESTRUCTURING CLASS HIERARCHIES

7.1 Students and Professors Reconsidered

Table III shows the final table for the example of Figure 1, as obtained by analyzing the class hierarchy along with the two example programs. The lattice corresponding to this table was shown previously in Figure 2 (note that we replaced member definitions by the corresponding method names there for convenience).

The following can be learned from the lattice:

—Data members that are not accessed anywhere in the program (e.g., `Person::socialSecurityNumber`) appear at the bottom element of the lattice.

—Data members of a base class B that are not used by (instances of) all derived classes of B are revealed. Such data members (e.g., `Person::address`) appear above (variables of) some but not all derived classes of B. For example, `Person::address` appears above instances of `Student`, but not above any instances of `Professor`.

—Variables from which no members are accessed appear at the the top element of the lattice (e.g., `s`).

—Data members that are properly initialized appear above the (constructor) method that is supposed to initialize them. If this is not the case, the data member may not be initialized. For example, we know that `Student::Student` does not initialize `Student::advisor` because that data member does not appear above `Student::Student` in the lattice.

—Situations where instances of a given type C access different subsets of C's members are revealed by the fact that variables of type C appear at different points in the lattice. Our example contains two examples of this phenomenon. The instances `Professor1` and `Professor2` of type `Professor` and the instances `Student1` and `Student2` of type `Student`.

As we mentioned earlier, a class hierarchy may be analyzed along with any number of programs, or without any program at all. The latter case may provide insights into the "internal structure" of a class library. Figure 9 shows the lattice obtained by analyzing the class hierarchy of Figure 1(a) *without* the programs of Figures 1(b) and (c); only code in method bodies is analyzed. Clearly, the resulting lattice should not be interpreted as a restructuring proposal, because it does not reflect the *usage* of the class hierarchy. However, there are some interesting things to note. For example, `socialSecurityNumber` is not accessed anywhere. If we would know in addition that `socialSecurityNumber` is `private` (i.e., that it can only be accessed by methods within its class), we could inform the user that it is effectively dead. Observe also that no members are accessed from method parameters `s` and `p`. Since the scope of these variables is local to the library, we know that analyzing additional code will not change this situation.

7.2 Restructuring Transformations

Once the lattice has been displayed, it can help in understanding the actual behavior of a class hierarchy and thus serve as a basis for restructuring tasks (see Section 9). In addition, several global restructuring transformations are possible:

—Unlabeled ("empty") lattice elements correspond to classes without members and without variables using them. The lattice can be simplified by pruning all such elements, and directly connecting their subordinate and superordinate neighbors. The resulting structure is not a lattice anymore, but only a partial order, but this is not so important: a class hierarchy need only be a partial order, and lookup behavior and subobject selection are not affected.

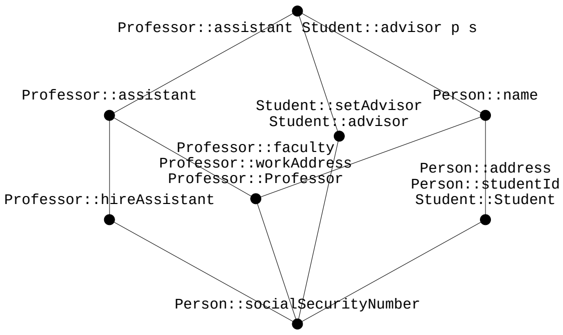


Fig. 9. Lattice obtained by analyzing the class hierarchy of Figure 1 without accompanying programs.

—A reduced lattice can be shown which contains only real objects, but not pointers. This lattice is obtained by deleting *all* pointer rows (not just `this`-pointers) from the final table, and it was explained earlier that the resulting lattice is a sublattice of the original one, and still operationally equivalent. For reengineering purposes, this lattice seems more appropriate than the fine-grained one: ultimately, objects access members and determine the optimal class structure; fine-grained behavior of pointers is generally not helpful in providing the overall picture.

—The user can decide to merge adjacent lattice elements if the distinction between these concepts is irrelevant. For example, one may decide that the distinction between between professors that hire assistants, and professors that do not hire assistants is irrelevant, and therefore merge the concepts for `Professor1` and `Professor2`. However, merging must respect the dominance constraints for members in order to to preserve member lookup behavior. For example, merging two concepts that have different definitions of a virtual method f associated with them is not possible, because at most one f can occur in any given class.

—With certain limitations, the user may move attributes upward in the lattice, and object downward. For example, the user may decide that `socialSecurityNumber` should be retained in the restructured class hierarchy, and move the corresponding attribute up to the concept labeled with attribute `Person::name`. Again, dominance constraints must be respected.

—Background knowledge that is not reflected in the lattice, e.g., "the type of x must be a base class of the type of y," can be integrated via background implications. Technically, background implications are treated the same way as dominance implications.

—Color should be used to display relevant substructures in the lattice, e.g., variables that formerly had the same type, or members that were formerly in the same class.

—Associations in the sense of UML can be recovered from the occurrences of class-typed members, and corresponding arcs added to the lattice.

—For very large class hierarchies, the tool could allow the user to focus on a selected subhierarchy either by specifying its minimal and maximal elements in the lattice, or by selecting specific rows and columns in the table (e.g., those belonging to a specific class).

—The structure theory of concept lattices offers several algebraic decompositions, such as horizontal decomposition, interference analysis, or block relations [Ganter and Wille 1999]. They can be used to measure quality factors such as cohesion and coupling [Snelting 1998; Lindig and Snelting 1997].

—Eventually, the user may associate names with lattice elements, which could be used as class names in the restructured hierarchy. For example, the programmer may determine that `Student` objects on which the `setAdvisor` method is invoked are graduate students, whereas `Student` objects on which this method is not called are undergraduates. Consequently, he may decide to associate names `Student` and `GraduateStudent` with the concepts labeled `s2` and `s1`, respectively.

—Finally, source code can be generated according to the new hierarchy, thereby utilizing the reduced object memory requirements and the improved structure in the new hierarchy.

7.3 Dealing with Multiple Inheritance

The analysis results are presented in form of a lattice, hence will naturally contain multiple inheritance if interpreted as a class hierarchy. Since Java does not support multiple inheritance, generated hierarchies may not be representable in Java source code. Note that if the meet point and its superclasses are in fact interface classes, there is no representation problem. Note further that multiple inheritance is only a problem if the method is to be used as a program transformation, but not if the lattice only serves program understanding.

Introducing a certain loss of precision, multiple inheritance can be removed as follows. Every occurrence of multiple inheritance leads to a “diamond” structure in the lattice, such as the diamond $Professor2 - Professor1 - p2 - Professor :: assistant$ in Figure 2. By moving members up and variables down (as explained above) the diamond can be transformed into a simple chain, while still maintaining behavioral equivalence. In Figure 2, *p2* can be moved down to *Professor2*, while $Professor ::$ $hireAssistant$ can be moved up to $Professor :: assistant$. Finally the lattice element formerly labeled *p2* can be removed, since it has become empty.

8. IMPLEMENTATION

A prototype implementation of the method was recently completed. Our tool is named KABA,[13] is written in Java and analyzes Java Class files. This approach has the advantage that no front end is needed. Furthermore, Java is much easier to analyze than C++.

8.1 CFG and Points-to Analysis

The tool first reads the required class files and builds a control flow graph (CFG). Since Java byte code is stack-oriented, but our analysis needs full variable references rather than anonymous stack entries, a simple backward analysis reconstructs the stack contents whenever necessary. If, at a certain point in the CFG, we need to know the type of, for example, the third entry from the top of stack, we explore all CFG paths backward until three push operations have been encountered on every

[13]KABA = KlassenAnalyse mit BegriffsAnalyse [class analysis using concept analysis]. KABA is also a popular chocolate drink in Germany.

backward path, and collect the items pushed onto the stack by the third-last push operations. Usually, the resulting sets are unique.

For points-to analysis, we use Andersen's method, as described in Shapiro and Horwitz [1997]. This method is quite precise, but also expensive: it has worst-case time complexity $O(n^3)$ and is very space-intensive in practice. In fact, the points-to analysis turns out to be the bottleneck of the analysis.

Points-to analysis has originally been designed for imperative languages such as C, and we had to extend it for object-oriented languages. In particular, the treatment of virtual dispatch requires special attention. For both Andersen's and Steensgaard's method, the details are described in Streckenbach and Snelting [2000]. Roughly, dynamic dispatch is modeled as follows. Whenever a method call $o.m(x)$ is encountered during interprocedural iteration, the following steps are performed:

(1) let $o \mapsto \{o_1, \ldots, o_n\}$ and $x \mapsto \{x_1, \ldots, x_m\}$ be the points-to information for o and x as obtained so far (encoded in the points-to graph). Static lookup is used to resolve any of the calls $o_1.m(x), \ldots, o_n.m(x)$.

(2) Let $C_1{::}m(a_1), \ldots, C_n{::}m(a_n)$ be the methods identified by static lookup. For any a_i, x_j, add edges as required by the assignments $a_i = x_j$ to the points-to graph. Furthermore, add edges as required by the implicit assignment to the `this`-pointer $C_i{::}m = o_i$.

(3) In case m's return type is a class, let $r_1, \ldots, r_n$ be variables representing the return values of the $C_i{::}m(a_i)$ inside the method. For any assignment or similar use of the return value, such as in $y = o.m(x)$, add edges as required by the assignment $y = r_i$ to the points-to graph.

(4) Continue propagation of points-to information.

8.2 Generation of Table Entries and Implication Propagation

The table is implemented as a list of bit strings, and once points-to analysis has converged, entering the member access entries into the table is straightforward. Next, the assignment and dominance implications are extracted. Extracting the dominance rules is quite expensive, because for any classes $A \leq B$ and any columns $A{::}m, B{::}m$, every row must be checked for a double entry $(x, A{::}m)$ and $(x, B{::}m)$.

Assignment implications as well as dominance implications are arranged into directed graphs. While the assignment graph may contain cycles, the dominance graph cannot, as dominance edges always go from members in "lower" classes to members in "upper" classes (and the original class hierarchy of course is cycle free). Note that, while the set of assignment implications never changes, new dominance implications might be generated after applying assignment implications. Hence, the dominance graph can grow during implication propagation. This is the reason why the $O(n^2)$ method for applying implications to a table [Ganter and Wille 1999] cannot be used, and a fixpoint iteration must be used instead.

Assignment and dominance implications are applied alternatively. "Local" iteration applies assignment implications respectively dominance implications until they converge. "Global" iteration alternates local assignment or dominance iterations. Implication propagation proceeds in topological order, and never propagates out of a cycle until the cycle converges. Since the dominance graph is cycle free, the corresponding local iteration converges immediately.

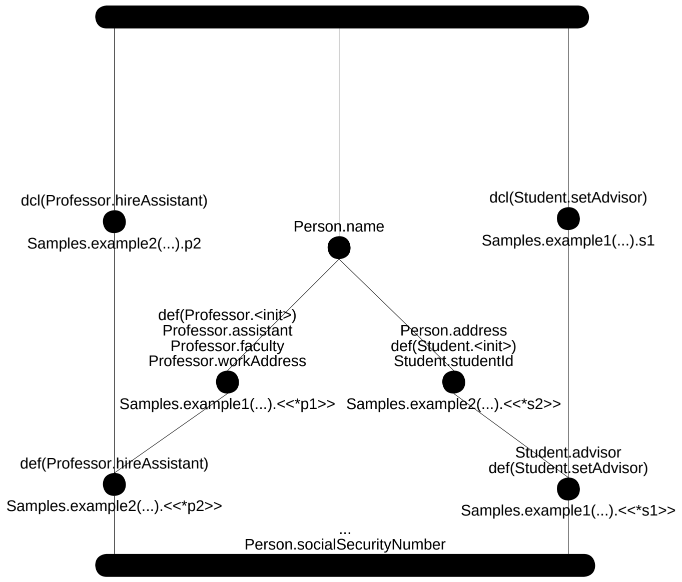


Fig. 10. Java version of student/professor example.

8.3 Interactive Back End

From the final table, the lattice is computed using Ganter's algorithm. Next, an off-the-shelf graph layouter is used to compute an initial layout for the lattice. The lattice is displayed by an interactive back end.

The lattice layout may be modified manually, while the system maintains lattice integrity. There are several options for the display of lattice elements labels, namely no labels at all, individual labels on request, and labels for user-defined entities only. The KABA prototype also offers some of the reengineering transformations that were discussed in Section 7, namely removal of empty lattice elements, reduced lattices without pointers, highlighting of variables that had the same original type, and recovery of associations. Interactive lattice manipulation and code generation are not supported yet.

9. CASE STUDIES

9.1 Students and Professors, Finally

KABA was applied to several small and medium-sized Java programs. We begin with a reconsideration of the student-professor example (see Figure 1), in order to illustrate differences between C++ and Java. Figure 10 presents a screenshot. Data members appear with their fully qualified name, whereas method definitions are distinguished from method declarations. Names in "`<<...>>`" are names of "real" objects (heap allocation sites), and constructor methods are named "`<init>`". Methods are displayed with full name, but without signature (signatures are shown only for overloaded methods). Top and bottom element are enlarged for layout reasons.

The first observation is that this lattice is different from the one in Figure 2. This is not a bug: The screenshot displays an analysis of the Java version of

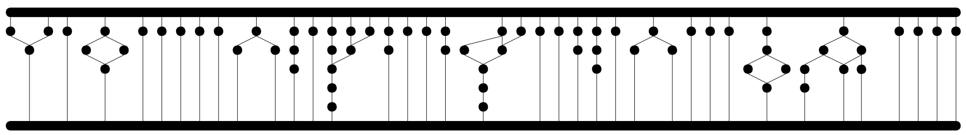


Fig. 11. Lattice for graph editor program.

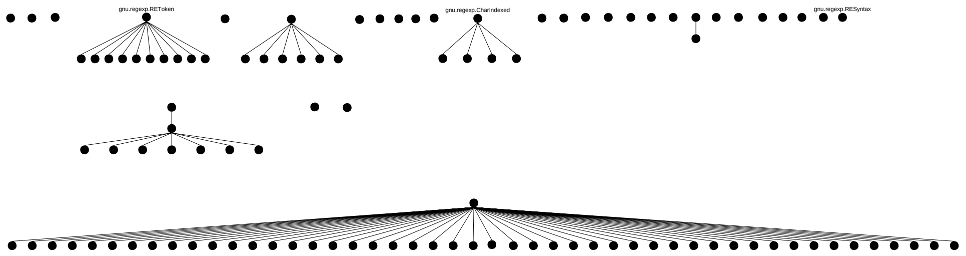


Fig. 12. Original class hierarchy for "jEdit" program.

Figure 1. In Java, all methods are virtual, while in Figure 1, `setAdvisor` and `hireAssistant` are nonvirtual. According to the table construction rules, receivers of virtual methods only need to see the method declaration; hence the definition of `Professor::hireAssistant` need not be visible to `p2`. Consequently, data member `Professor::assistant` need not be visible to `p2`. Therefore a corresponding table entry is not created, and the distinction between the two rightmost lattice elements in Figure 2 disappears. The same argument applies to `Student::advisor` except that the original lattice did not contain such a distinction anyway (due to the missing initialization of `Student::advisor` in the constructor; see Section 7). As a result of this subtle phenomenon, the Java version of the lattice is completely symmetrical, indicating the lower semantic complexity of Java vs. C++.

9.2 An Easy Case

Our next example is a graph editor program (3761 LOC \ comments) with a completely flat class structure. The purpose of this experiment was to see whether KABA proposes to introduce inheritance and specialized subclasses.

The lattice (Figure 11) is horizontally decomposable into several small sublattices (actually, each sublattice corresponds to one original class). The internal structure in the sublattices stems from fine-grained pointer access patterns and should not be interpreted as an option to split classes. In particular, each substructure in the lattice (except two) has its own local bottom element, which is alway an indicator that potential for introducing inheritance and splitting classes is low. Indeed, the reduced lattice without pointers replicates the original hierarchy. Thus KABA demonstrates that there is no refactoring potential. This example shows that our approach is useful not only for reengineering, but also for ongoing quality assurance during development: It can confirm that the class design corresponds to actual class usage, and hence that refactoring is not necessary.

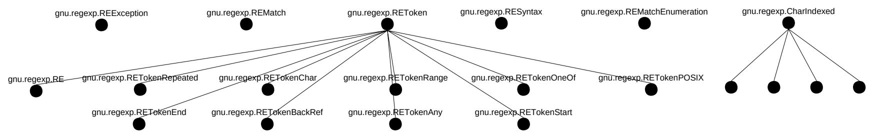
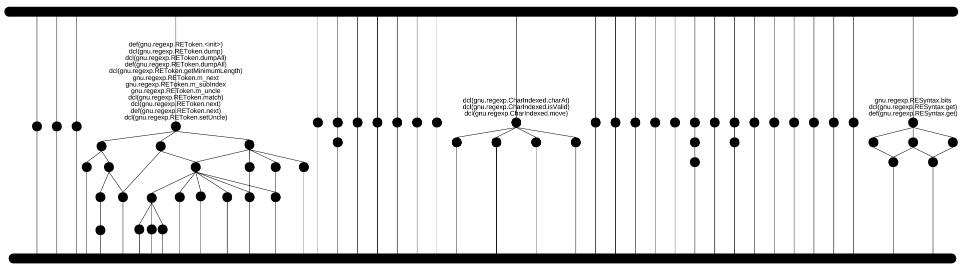
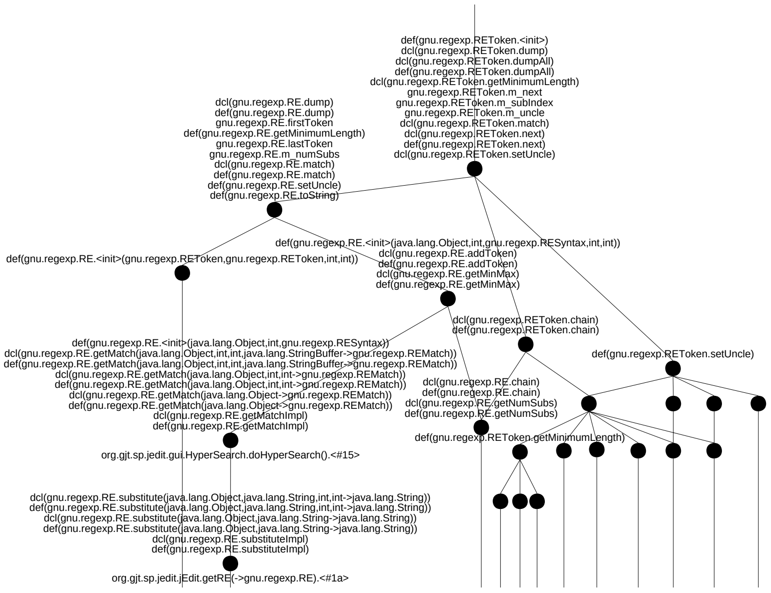


Fig. 13. Original subhierarchy for the regular expression library in “jEdit”.



def(gnu.regexp.REToken.<init>)
dcl(gnu.regexp.REToken.dump)
dcl(gnu.regexp.REToken.dumpAll)
def(gnu.regexp.REToken.dumpAll)
dcl(gnu.regexp.REToken.getMinimumLength)
gnu.regexp.REToken.m_next
gnu.regexp.REToken.m_subIndex
gnu.regexp.REToken.m_uncle
dcl(gnu.regexp.REToken.match)
dcl(gnu.regexp.REToken.next)
def(gnu.regexp.REToken.next)
dcl(gnu.regexp.REToken.setUncle)
dcl(gnu.regexp.CharIndexed.charAt)
dcl(gnu.regexp.CharIndexed.isValid)
dcl(gnu.regexp.CharIndexed.move)
gnu.regexp.RESyntax.bits
dcl(gnu.regexp.RESyntax.get)
def(gnu.regexp.RESyntax.get)

Fig. 14. Lattice for “jEdit”.



def(gnu.regexp.REToken.<init>)
dcl(gnu.regexp.REToken.dump)
dcl(gnu.regexp.REToken.dumpAll)
def(gnu.regexp.REToken.dumpAll)
dcl(gnu.regexp.REToken.getMinimumLength)
gnu.regexp.REToken.m_next
gnu.regexp.REToken.m_subIndex
gnu.regexp.REToken.m_uncle
dcl(gnu.regexp.REToken.match)
dcl(gnu.regexp.REToken.next)
def(gnu.regexp.REToken.next)
dcl(gnu.regexp.REToken.setUncle)
dcl(gnu.regexp.RE.dump)
def(gnu.regexp.RE.dump)
gnu.regexp.RE.firstToken
def(gnu.regexp.RE.getMinimumLength)
gnu.regexp.RE.lastToken
gnu.regexp.RE.m_numSubs
dcl(gnu.regexp.RE.match)
def(gnu.regexp.RE.match)
def(gnu.regexp.RE.setUncle)
def(gnu.regexp.RE.toString)
def(gnu.regexp.RE.<init>(gnu.regexp.REToken,gnu.regexp.REToken,int,int))
def(gnu.regexp.RE.<init>(java.lang.Object,int,gnu.regexp.RESyntax,int,int))
dcl(gnu.regexp.RE.addToken)
def(gnu.regexp.RE.addToken)
dcl(gnu.regexp.RE.getMinMax)
def(gnu.regexp.RE.getMinMax)
dcl(gnu.regexp.REToken.chain)
def(gnu.regexp.REToken.chain)
def(gnu.regexp.REToken.setUncle)
def(gnu.regexp.RE.<init>(java.lang.Object,int,gnu.regexp.RESyntax))
dcl(gnu.regexp.RE.getMatch(java.lang.Object,int,int,java.lang.StringBuffer->gnu.regexp.REMatch))
def(gnu.regexp.RE.getMatch(java.lang.Object,int,int,java.lang.StringBuffer->gnu.regexp.REMatch))
dcl(gnu.regexp.RE.getMatch(java.lang.Object,int,int->gnu.regexp.REMatch))
def(gnu.regexp.RE.getMatch(java.lang.Object,int,int->gnu.regexp.REMatch))
dcl(gnu.regexp.RE.getMatch(java.lang.Object->gnu.regexp.REMatch))
def(gnu.regexp.RE.getMatch(java.lang.Object->gnu.regexp.REMatch))
dcl(gnu.regexp.RE.getMatchImpl)
def(gnu.regexp.RE.getMatchImpl)
org.gjt.sp.jedit.gui.HyperSearch.doHyperSearch().<#15>
dcl(gnu.regexp.RE.chain)
def(gnu.regexp.RE.chain)
dcl(gnu.regexp.RE.getNumSubs)
def(gnu.regexp.RE.getNumSubs)
def(gnu.regexp.REToken.getMinimumLength)
dcl(gnu.regexp.RE.substitute(java.lang.Object,java.lang.String,int,int->java.lang.String))
def(gnu.regexp.RE.substitute(java.lang.Object,java.lang.String,int,int->java.lang.String))
dcl(gnu.regexp.RE.substitute(java.lang.Object,java.lang.String->java.lang.String))
def(gnu.regexp.RE.substitute(java.lang.Object,java.lang.String->java.lang.String))
dcl(gnu.regexp.RE.substituteImpl)
def(gnu.regexp.RE.substituteImpl)
org.gjt.sp.jedit.jEdit.getRE(->gnu.regexp.RE).<#1a>

Fig. 15. Details for “jEdit”: regular expression classes.

9.3 The GNU Regular Expression Library

Our next example is “jEdit,” a text editor with useful features such as syntax coloring and regular expression search.[14] JEdit contains more than 80 classes and almost 12,000 LOC. It makes heavy use of a Java adaptation of the GNU regular expression library, and can thus be seen as an instance of our scenario, namely that

[14]version 1.2final, available from `http://www.gjt.org/s̃p/jedit.html`

a given hierarchy is used by different applications.

Figure 12 shows the original hierarchy of all classes shipped with "jEdit." Five separate subsystems are visible, concerned with input modes, editor commands, editor modes, regular expressions, and syntax highlighting. Several singleton classes without any inheritance relationship provide basic and auxiliary functionality. All original subhierarchies are very flat.

Figure 13 shows those classes of the original hierarchy which constitute the regular expression library; these are 20 classes comprising more than 3000 LOC. The superclass `REToken` has one subclass for every regular expression construct $(*, +, [\ ], \$)$. The programming interface class `RE` also is a subclass of `REToken`.

Figure 14 shows the reduced lattice produced by KABA.[15] It shows several independent substructures that correspond to the subsystems of the original hierarchy. Most of the singleton classes in the original hierarchy, as well as the entire "input mode" subsystem, recur literally in the right-hand part of the lattice, indicating low reengineering potential.

More interestingly, however, is the leftmost part of the lattice, which represents the regular expressions library (that is, all subclasses of `REToken`), which has become a complex structure. Figure 15 shows a detailed view. The rightmost part of this structure represents the classes for the different regular expression constructs. Note that there is more fine-grained detail than before because the different constructs apparently rely on different parts of `REToken`'s functionality. It is interesting to observe that while some subclasses of the original base class `REToken` are literally reproduced by KABA, the original subclass `RE` has been distributed over 6 different nodes (left part of Figure 15). A look at the source code reveals that the class labeled `gui.HyperSearch.doHyperSearch` is the API for search without substitution, whereas the class below it is the API for search with substitution.

Note that the lattice displays the finest possible splittings and refactorings of classes according to possible program behavior. For reengineering purposes, the lattice should therefore be simplified by merging lattice elements, in order to reflect software design principles. Nevertheless, the lattice demonstrates that the original `RE` class can be split into, say, `RE_substitution` and `RE_no_substitution`. The element labeled `gnu.regexp.RE.dump` etc. also reveals that there is a "composite" design pattern used: class `RE` stands for complex regular expression and offers methods for accessing subexpressions, while the subclasses of `REToken` in the right part represent elementary regular expressions.

9.4 JAS

Our next example is "JAS", a java bytecode assembler, including a Scheme-like scripting language (about 5400 LOC).[16] Its original class hierarchy is shown in Figure 16. Among various single classes and three small inheritance trees it shows a huge structure with more than 50 classes. These classes are part of the scripting

[15]For this and the following example we used the reduced lattice which does not show fine-grained access patterns for pointers, just for "real" objects. As explained in Section 7, fine-grained pointer access patterns are not really relevant for reengineering, and the reduced lattice is still guaranteed to be operationally equivalent. The full lattices for this and the next example are about twice the size as the reduced lattices.

[16]version 0.4, available from `http://www.sbktech.org/jas.html`

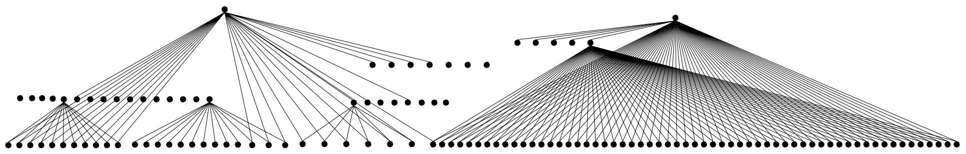


Fig. 16. Original class hierarchy for the “JAS” example.

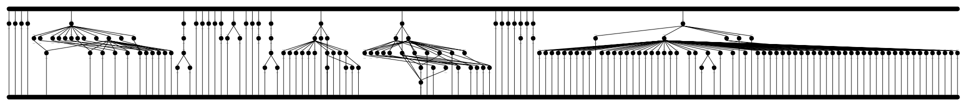


Fig. 17. Lattice for “JAS”.

language implementation. The top class is called `Obj`, and all (except four) subclasses additionally implement an interface `Procedure`. Each of these subclasses represents a function like “Add” or “Sub” in the scripting language.

In the KABA lattice (Figure 17) this huge structure is reproduced basically unmodified, confirming the original design was good. However, the subhierarchy with base class `Insn` (Figure 18) is interesting. All but one subclasses of `Insn` just redefine the constructor method, and these subclasses are reproduced unchanged. However, the rightmost chain in Figure 18, which contains all members of the original subclass `Label`, differs from the other subclasses because it does not use the methods `Insn.size` and `Insn.write`. A closer look reveals that all the other subclasses are dealing with the implementation of certain bytecode instructions, but `Label` is concerned with bytecode addressing. The implementations of `size` and `write` in `Label` are empty, so these two methods can be considered “amputated.” An even closer look reveals that the `resolv` method does not execute any useful code when called from a `Label` object. This demonstrates that the original subhierarchy should be restructured: `Label` does not share any code with the other subclasses; thus it does not need a common base class with them.

The sublattice for the subclasses of the `InsnOperand` class shows a similar phenomenon (Figure 19). Two classes (`UnsignedByteWideOperand` and `IincOperand` on the left-hand side) are separated from the rest, just like `Label` was. They have their own implementations of the method `writePrefix`, while all other subclasses share the same implementation. A look at the source code reveals that the other subclasses use a dummy implementation of `writePrefix` which has no functionality; only the two “separated” classes on the left actually have code for `writePrefix`. This demonstrates that `writePrefix` can be removed from `InsnOperand` and put into a new class, which should be the base class of the separated classes.

9.5 Empirical Results

Some additional case studies are described in the Bögemann/Streckenbach master’s thesis [Bögemann and Streckenbach 1999]. Of course, it is interesting to know about the performance and lattice size as a function of program size. Table IV presents data for the above case studies and for KABA self-application. The number of rows in a KABA-generated table is roughly the sum of the number of pointers plus the

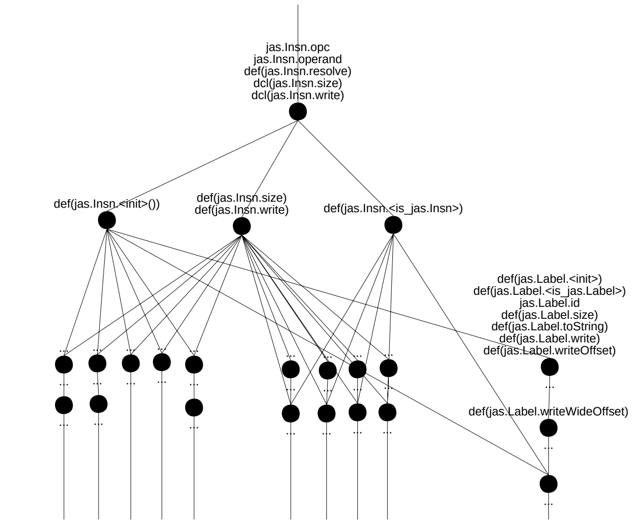


Fig. 18. Details for "JAS": substructure for `Insn`.

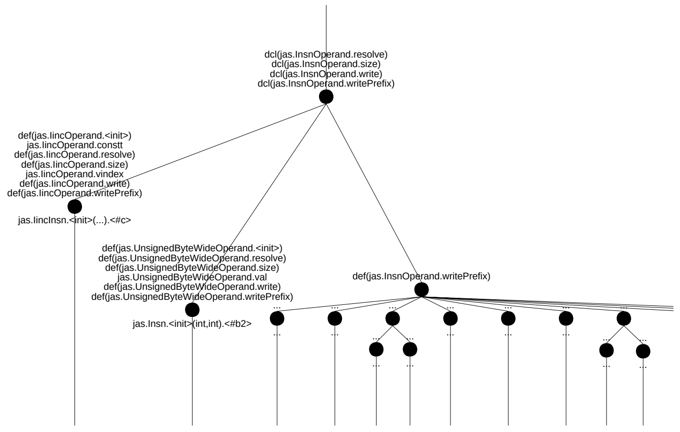

dcl(jas.InsnOperand.resolve)
dcl(jas.InsnOperand.size)
dcl(jas.InsnOperand.write)
dcl(jas.InsnOperand.writePrefix)
def(jas.IincOperand.<init>)
jas.IincOperand.constt
def(jas.IincOperand.resolve)
def(jas.IincOperand.size)
jas.IincOperand.vindex
def(jas.IincOperand.write)
def(jas.IincOperand.writePrefix)
jas.IincInsn.<init>(...).<#c>
def(jas.UnsignedByteWideOperand.<init>)
def(jas.UnsignedByteWideOperand.resolve)
def(jas.UnsignedByteWideOperand.size)
jas.UnsignedByteWideOperand.val
def(jas.UnsignedByteWideOperand.write)
def(jas.UnsignedByteWideOperand.writePrefix)
jas.Insn.<init>(int,int).<#b2>
def(jas.InsnOperand.writePrefix)

Fig. 19. Details for "JAS": substructure for `InsnOperand`.

number of objects (creation sites); usually there are 10 times as many pointers as creation sites. The number of columns includes members/methods from standard library usage; it is about one-third higher than the total number of members and methods (due to the def/decl distinction for methods). Note the high number of assignment implications (including implicit assignments for parameters, `this` pointers, etc.). The number of dominance implications is comparatively small, and on the average only 25% of them ever generate a new table entry. The biggest program, KABA itself, is 26,882 lines long and has a table with about $5.43 \cdot 10^8$ cells; only a small percentage of these have an entry. Note the high number of assignment implications for KABA.

In order to eliminate fine-grained pointer access patterns, all rows for pointers

Table IV. Program and Table Statistics for Case Studies. The columns display lines of code, total number of pointers, total number of objects, number of table rows, number of table columns, number of assignment implications, number of dominance implications.

	Program Statistics			Table Statistics			
	LOC	#Ptr	#Obj	#Row	#Col	#AssImp	#DomImp
graph	7045	8518	678	9197	1137	16399	1098
jEdit	11862	18597	1877	20477	2881	62508	3245
jas	5432	12590	2127	14718	1457	72735	1875
KABA	26882	73842	5722	79580	6831	648494	7478

Table V. Performance and Lattice Statistics for Case Studies. The columns display times for points-to-analysis, table construction including implication iteration, and construction of the final lattice (all measured on a SUN 450); furthermore number of rows and columns in the final table and number of elements in the final lattice are displayed.

	Performance Statistics			Lattice Statistics		
	Points-to	Table	Lattice	#finRow	#finCol	#LatElem
graph	48.28 sec	24.3 sec	0.00 sec	106	335	57
jEdit	514.84 sec	260.77 sec	0.30 sec	211	801	136
jas	660.15 sec	80.56 sec	0.20 sec	527	771	140
KABA	12577.98 sec	2355.48 sec	0.38 sec	1147	3074	519

are deleted after table construction (see Section 7.2). Furthermore, all columns for methods from the standard libraries are deleted as well. This dramatically reduces the table and lattice size: the final table for KABA has only $3.52 \cdot 10^6$ cells, which is less than one percent of the full table. The lattice generation is based on the final tables; it uses the very efficient implementation of Ganter's algorithm by Christian Lindig [Lindig 1999], and the times are neglectable. We mention in passing that the KABA self-application lattice is quite big, but very flat and very similar to the original class hierarchy.

Table V summarizes execution times for the case studies. Obviously, program size does not correlate well with points-to analysis performance. The extremely high points-to time for KABA is due to excessive garbage collection and paging; the other programs need less memory for the points-to graph, which leads to more realistic points-to times in the 10-minute range. The times for table construction and implication propagation are much less than the points-to times.

Obviously, the initial points-to analysis is the bottleneck of the whole method. Worse, an implementation of the supposedly faster (and less precise) Steensgaard algorithm, adapted for Java, was not faster in practice: the imprecision propagates and eventually slows down convergence of the iteration for the conservative approximation of dynamic binding. Other authors also have observed that the speed of Steensgaard's analysis is offset by increased costs for later client analyses [Bent et al. 2000; Hind and Pioli 2000]. We agree with Bent et al. [2000] that a flow- and context-sensitive points-to analysis could well be worth the effort.

We expect that a native code compiler and a better garbage collector will reduce time and memory requirements by at least 30%. However, the ultimate improvement must come from the treatment of standard libraries. KABA currently performs a whole-program analysis and conservatively approximates the effect of library calls. It would certainly be better if fine-grained information is precomputed for library functions, as in Rountev and Ryder [2000].

The preliminary experience from our experiments can be summarized as follows:

—Most Java programs explicated a reasonable structure without high reengineering potential, probably due to the fact that these programs are quite young. In such cases, the lattice can serve as a quality metric, demonstrating that the architecture is good.

—Nevertheless, in many Java examples we found possibilities to split or refactor classes. These proposals, which are guaranteed not to alter the program behavior, would not have been possible without our unique combination of points-to analysis, type constraints, and concept lattices.

—KABA has an experimental option where member accesses from dead code will not be entered into the table. This greatly reduces the size of the lattices in many examples. From a reengineering viewpoint however, it is questionable to exclude dead code, just as it is questionable to delete dead members such as `socialSecurityNumber` in our first example.

—We did not yet exploit the structure theory of concept lattices, in particular congruences and weak congruences. (Weak) congruence classes could serve as proposals how to group classes into packages, and can be used to measure coupling and cohesion; resulting in more substantial restructuring proposals.

—The full set of reengineering transformations from Section 7 is not available yet.

—The real market for our method consists of course of ill-structured legacy C++ applications with a long revision history. Given the complexity of both C++ and our method, we think it is realistic to say that the application of our techniques to legacy C++ applications is at least a few years of work.

10. RELATED WORK

10.1 Applications of Concept Analysis

Godin and Mili [Godin and Mili 1993; Godin et al. 1998] also use concept analysis for class hierarchy (re)design. The starting point in their approach is a set of interfaces of (collection) classes. A table is constructed that specifies for each interface the set of supported methods. The lattice derived from this table suggests how the design of a class hierarchy implementing these interfaces could be organized in a way that optimizes the distribution of methods over the hierarchy. Although Godin and Mili's work has the same formal basis as ours, the domains under consideration are different. In Godin and Mili [1993], relations between members and classes are studied in order to improve the distribution of these members over the class hierarchy. In contrast, we study how the members of a class hierarchy are *used* in the executable code of a set of applications by examining relationships between variables and class members, and relationships among class members.

Another application of concept analysis in the domain of software engineering is the analysis of software configurations. Snelting [1996] uses concept analysis to analyze systems in which the C preprocessor (CPP) is used for configuration management. The relation between code pieces and governing expressions is extracted from a source file, and the corresponding lattice visualizes interferences between configurations. Later, Lindig proved that the configuration space itself is isomorphic to the lattice of the *inverted* relation [Lindig 1998].

Concept analysis was also used for modularization of old software. Siff and Reps [1997] investigated the relation between procedures and "features" such as usage of global variables or types. A modularization is achieved by finding elements in the lattice whose intent partitions the feature space. Lindig and Snelting [1997] also analyzed the relation between procedures and global variables in legacy Fortran programs. They showed that the presence of module candidates corresponds to certain decomposition properties of the lattice (the Siff/Reps criterion being a special case).

Ball [1999] applied concept analysis to test coverage information. In his domain, the objects are tests and attributes are the program entities (e.g., procedures) that a test may cover. The concept lattices derived from these tables identify dynamic control flow invariants between entities that are similar to static program properties such as domination and postdomination.

10.2 Class Hierarchy Specialization and Application Extraction

The work in the present article is closely related to the work on *class hierarchy specialization* by Tip and Sweeney [1997; 2000]. Class hierarchy specialization is a space optimization technique in which a class hierarchy and a client program are transformed in such a way that the client's space requirements are reduced at run-time. The method of Tip and Sweeney [1997; 2000] shares some basic "information gathering" steps with the method of the present article,[17] but its subsequent steps are quite different. After determining the member access and assignment operations in the program, a set of *type constraints* is computed that capture the subtype-relationships between variables and members that must be retained. These type constraints roughly correspond to the information encoded in our tables, but contrary to our current approach they correctly distinguish between multiple subobjects that have the same type. From the type constraints, a new class hierarchy is generated automatically. In a separate step, the resulting class hierarchy is simplified by repeatedly applying a set of simple graph rewriting rules.

In addition to the differences in the underlying algorithms, the method of Tip and Sweeney [1997; 2000] differs from our reengineering framework in a number of ways. Class hierarchy specialization is an optimization technique that does not require any intervention by the user. In contrast, the current article presents an *interactive* approach for analyzing the usage of a class hierarchy in order to find design problems. Reducing object size through the elimination of members is possible, but not necessarily an objective. For the purpose of restructuring it may very well be the case that an unused member should be retained in the restructured class hierarchy. The framework we presented here also allows for the analysis of a class hierarchy along with any number of programs, including none. Class hi-

[17]Definitions 1, 3, 4, and 7 were taken from Tip and Sweeney [1997; 2000].

erarchy specialization customizes a class hierarchy with respect to a *single* client application.

Several other *application extraction* techniques for eliminating unused components from hierarchies and objects have been presented in the literature. These are primarily intended as optimizations, although they may have some value for program understanding. Agesen and Ungar [1994] describe an algorithm for the dynamically typed language Self that eliminates unused slots from objects (a slot corresponds to either a data member, a method, or an inheritance relation). Self is a dynamically typed language, and eliminating members from objects does not involve transforming class hierarchies.

Tip et al. [1996] present an algorithm for slicing class hierarchies that eliminates members and inheritance relations from a C++ hierarchy. Class slicing is less powerful than specialization because it can only remove a member m from a class C if m is not used by *any* C-instance. Later, Tip et al. developed *Jax* [Tip et al. 1999], an application extractor for Java, which incorporates Rapid Type Analysis [Bacon and Sweeney 1996] to construct call graphs and detect unreached methods, elimination of dead fields [Sweeney and Tip 1998], as well as some of the class hierarchy transformations of Tip and Sweeney [1997; 2000]. *Jax* reduces the size of class file archives by up to 70%.

10.3 Techniques for Restructuring Class Hierarchies

Another category of related work is that of techniques for restructuring class hierarchies for the sake of improving design and reuse. The overview article by Casais [1998] presents 18 different methods, many of them process-centered or dynamic analyses. The probably most well-known method for static restructuring was introduced by Opdyke and Johnson [Opdyke 1992; Opdyke and Johnson 1993]. They present a number of behavior-preserving transformations on class hierarchies, which they refer to as *refactorings*. The goal of refactoring is to improve design and enable reuse by "factoring out" common abstractions. This involves steps such as the creation of new superclasses, moving around methods and classes in a hierarchy, and a number of similar steps. Our techniques for analyzing the usage of a class hierarchy to find design problems is in our opinion complimentary to the techniques of Opdyke [1992] and Opdyke and Johnson [1993].

Moore [1996] presents a tool that automatically restructures inheritance hierarchies and refactors methods in Self programs. The goal of this restructuring is to maximize the sharing of expressions between methods, and the sharing of methods between objects in order to obtain smaller programs with improved code reuse. Since Moore is studying a dynamically typed language without explicit class definitions, a number of complex issues related to preserving the appropriate subtype-relationships between types of variables do not arise in his setting.

An interesting approach is that of Astudillo [1997]. He argues that from an evolutionary viewpoint, subclasses may not only add or redefine members, but also loose or "amputate" members. This approach violates fundamental type-theoretic properties of object-oriented programming, but has the advantage that well-known algorithms for the reconstruction of biological taxonomies can be used. Astudillo argues that his hierarchies are more "natural" than those which stick to the principles of type conformance and contravariance.

11. CONCLUSIONS AND FUTURE WORK

We have presented a method for understanding class hierarchies by analyzing the *usage* of the hierarchy by a set of applications. This method constructs a *concept lattice* in which relationships between variables and class members are made explicit, and where information that members and variables have in common is "factored out." We have shown the technique to be capable of finding design anomalies such as class members that are redundant or that can be moved into a derived class. In addition, situations where it is appropriate to split a class can be detected. We have suggested how these techniques can be incorporated into interactive tools for maintaining and restructuring class hierarchies.

Our analysis is one of the most powerful analysis methods for object-oriented programs, due to its unique combination of points-to analysis, type constraints, and concept lattices. The method subsumes classic analyses such as dead-member detection and useless-variable detection as special cases. Our preliminary case studies have indicated the usefulness of the analysis as a basis for reengineering, but the method can also be used for quality assessment during initial development. It turned out that the Java examples we analyzed were all reasonably well structured, but that the method nevertheless discovered many possibilities for refactoring, while at the same time guaranteeing that program behavior is unchanged.

11.1 Future Work

The present article has focused on foundational aspects and preliminary case studies. We distinguish the following avenues for future work:

Language Issues. Several important language features have not been discussed yet. Union types and parametric polymorphism pose interesting research problems. In the presence of union types, different sets of members may be accessed from a union's variants. A solution with distinct lattice elements for the different variants seems the obvious solution. In the case of parametric polymorphism, the question arises of what to do when two instantiations of a generic type can be restructured differently.

Interactive Restructuring Support. This article has focused primarily on providing programmers with information on how a class hierarchy is used. The obvious next step is to construct tools that allow for the interactive restructuring of class hierarchies based on the results of the analysis. Actions supported by such a tool could include:

—Deleting unlabeled lattice elements.
—Merging adjacent lattice elements in cases where certain fine distinctions are deemed irrelevant by the reengineer.
—Moving members up in the lattice and variables down in the lattice. The tool would check that no type constraints are violated by these actions.
—Incorporating background knowledge into a class hierarchy ("X must be a subclass of Y").
—Using color to highlight substructures. For example, all members that originated from the same source class could be shown in the same color.
—Utilizing algebraic decompositions and the structure theory of concept lattices.
—Associating names with lattice elements.

—Generating restructured class hierarchies and code from a lattice.

Developing and Utilizing more Precise Points-to Analyses. In Section 9.5, we reported problems with the current points-to analysis technology. In a nutshell, we found that Andersen's analysis [Andersen 1994] is very expensive on larger programs, and that the imprecision of Steensgaard's inexpensive points-to analysis [Steensgaard 1996] slows down KABA's approximation of dynamic binding in a way that offsets its speed. Das [2000] recently developed *one-level-flow* analysis, and reported precision results comparable with Andersen and analysis speed comparable with Steensgaard for most C programs. The applicability of Das' algorithm to Java remains to be seen, because of the absence of multilevel pointers, and the pervasive use of virtual method dispatch. In our opinion, the usefulness of unification-based points-to analysis algorithms such as Steensgaard's and Das' in the object-oriented domain is diminished by the fact that unification interferes with the subtyping relationships that are abundant in typical object-oriented programs.

In our experience, the conservative approximation of dynamic dispatch due to imprecise points-to relations has the dual disadvantages of both increasing the cost and reducing the accuracy of our subsequent analysis. This suggests the use of more precise flow- and context-sensitive points-to analysis algorithms. A more efficient treatment of library functions could also be worthwhile.

Industrial Application. Ill-structured legacy C++ applications with a long revision history are the real market for the method. It remains to be seen whether an efficient implementation for the full C++ language can be achieved.

APPENDIX

This appendix demonstrates that a method and its `this` pointer will always appear together in the lattice, and that arbitrary pointers appear above any object they point to.

LEMMA A.1. *For any $a \in \mathcal{A}$, $\mu(a) = \bigvee_{(o,\,a)\in T} \gamma(o)$.*

PROOF. This is true for every concept lattice by construction [Ganter and Wille 1999]. □

LEMMA A.2. *For any $\mathit{def}(C{::}f) \in \mathit{MemberDefs}(\mathcal{P})$, we have that*

$$\gamma(*C{::}f) \geq \mu(\mathit{def}(C{::}f)).$$

PROOF. By the preceding lemma,

$$\mu(\mathit{def}(C{::}f)) = \bigvee_{(x,\,\mathit{def}(C::f))\in T} \gamma(x).$$

Furthermore, for any x such that $(x, \mathit{def}(C{::}f)) \in T$, there must be a method call $x.f()$ (otherwise the table entry would not exist). This method call causes an implicit assignment to f's `this` pointer, generating an assignment dominance $*C{::}f \rightarrow x$, which enforces $\gamma(*C{::}f) \geq \gamma(x)$. Since this is true for all x calling f, it is also true for their supremum:

$$\gamma(*C{::}f) \geq \bigvee_{(x,\,\mathit{def}(C::f))\in T} \gamma(x)$$

Combining both statements, we obtain $\gamma(*C{::}f) \geq \mu(\mathit{def}(C{::}f))$. $\square$

The last lemma shows that a method always appears below its `this` pointer, and without the `this` rule, they will indeed appear at different elements in the lattice if method $C{::}f$ does not access itself (i.e., is nonrecursive).

The `this` rule enforces $\gamma(*C{::}f) \leq \mu(\mathit{def}(C{::}f))$, and together with the lemma we may conclude the following.

PROPOSITION A.3. *For any $\mathit{def}(C{::}m) \in \mathit{MemberDefs}(\mathcal{P})$ we have that*

$$\gamma(*C{::}f) = \mu(\mathit{def}(C{::}f)).$$

Hence methods and their `this` pointers appear together in the lattice. For pointers in general, only a weaker result can be established: any pointer always appears above any object it may point to.

PROPOSITION A.4. $\langle p, v\rangle \in \mathit{PointsTo}(\mathcal{P}) \implies \gamma(p) \geq \gamma(v)$.

PROOF. (In this proof, we use member m also as a shorthand for $\mathit{def}(X{::}m)$ or $\mathit{dcl}(X{::}m)$.) By the basic properties of concept lattices, it is enough to show

$$\forall m \in \mathit{MemberDcls}(\mathcal{P}) : (p, m) \in T \implies (v, m) \in T$$

because this implication will force $\gamma(p) \geq \gamma(v)$. So let $(p, m) \in T$. By Definition 5, this implies $\langle m, p\rangle \in \mathit{MemberAccess}(\mathcal{P})$ and therefore $p.m()$ must occur in $\mathcal{P}$. Since $\langle p, v\rangle \in \mathit{PointsTo}(\mathcal{P})$, by Definition 4 (case 3), $\langle m, v\rangle \in \mathit{MemberAccess}(\mathcal{P})$ and therefore $(v, m) \in T$. $\square$

ACKNOWLEDGMENTS

Andreas Bögemann and Mirko Streckenbach did a great job with the prototype implementation [Bögemann and Streckenbach 1999], and Mirko's help with the experiments and with improving the prototype was indispensable.

REFERENCES

ACCREDITED STANDARDS COMMITTEE X3. 1997. Working paper for draft proposed international standard for information systems—programming language C++. Doc. No. X3J16/97-0108.

AGESEN, O. AND UNGAR, D. 1994. Sifting out the gold: Delivering compact applications from an exploratory object-oriented programming environment. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*. Portland, OR, 355–370. *ACM SIGPLAN Notices* 29(10).

ANDERSEN, L. O. 1994. Program analysis and specialization for the c programming language. Ph.D. thesis, DIKU, University of Copenhagen. DIKU report 94/19.

ASTUDILLO, H. 1997. Maximizing object reuse with a biological metaphor. *Theory and practice of object systems 3,* 4, 235–251.

BACON, D. F. AND SWEENEY, P. F. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*. San Jose, CA, 324–341. *ACM SIGPLAN Notices* 31(10).

BALL, T. 1999. The concept of dynamic analysis. In *Proceedings of the Seventh European Software Engineering Conference Held Jointly with the Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering.* Toulouse, France, 216–234.

BENT, L., ATKINSON, D., AND GRISWOLD, W. G. 2000. A comparative study of two whole-program slicers for C. Tech. Rep. CS2000–643, UCSD. May.

Birkhoff, G. 1940. *Lattice Theory.* American Mathematical Society.

Bögemann, A. and Streckenbach, M. 1999. KABA: Reengineering class hierarchies using concept lattices. M.S. thesis, Technische Universität Braunschweig, Germany.

Casais, E. 1998. Reengineering of object-oriented legacy systems. *Journal of object-oriented programming*, 45–52.

Choi, J., Grove, D., Hind, M., and Sarkar, V. 1999. Efficient and precise modeling of exceptions for the analysis of java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '99).* ACM, Toulouse, France, 21–31.

Choi, J.-D., Burke, M., and Carini, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages.* ACM, 232–245.

Das, M. 2000. Unification-based pointer analysis with directional assignments. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00).* Vancouver, Canada, 35–46. Appeared as *ACM SIGPLAN Notices 35(5).*

Davey, B. and Priestley, H. 1990. *Introduction to lattices and order.* Cambridge University Press.

Ganter, B. and Wille, R. 1999. *Formal Concept Analysis - Mathematical Foundations.* Springer Verlag.

Godin, R. and Mili, H. 1993. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93).* Washington, DC, 394–410. *ACM SIGPLAN Notices* 28(10).

Godin, R., Mili, H., Mineau, G. W., Missaoui, R., Arfi, A., and Chau, T.-T. 1998. Design of class hierarchies based on concept (galois) lattices. *Theory and Practice of Object Systems 4,* 2, 117–134.

Hind, M. and Pioli, A. 2000. Which pointer analysis should I use? In *Proceedings of the International Symposium on Software testing and Analysis.* Portland. to appear.

Krone, M. and Snelting, G. 1994. On the inference of configuration structures from source code. In *Proceedings of the 1994 International Conference on Software Engineering (ICSE'94).* Sorrento, Italy, 49–57.

Lindig, C. 1998. Analyse von Softwarevarianten. Tech. Rep. 98-02, TU Braunschweig, FB Informatik.

Lindig, C. 1999. Algorithmen zur begriffsanalyse und ihre anwendung in softwarebibliotheken. Ph.D. thesis, Technische Universität Braunschweig. in German.

Lindig, C. and Snelting, G. 1997. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97).* Boston, MA, 349–359.

Moore, I. 1996. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96).* San Jose, CA, 235–250. *ACM SIGPLAN Notices* 31(10).

Opdyke, W. and Johnson, R. 1993. Creating abstract superclasses by refactoring. In *ACM 1993 Computer Science Conference.*

Opdyke, W. F. 1992. Refactoring object-oriented frameworks. Ph.D. thesis, University Of Illinois at Urbana-Champaign.

Pande, H. D. and Ryder, B. G. 1996. Data-flow-based virtual function resolution. In *Proceedings of the Third International Symposium on Static Analysis (SAS'96).* 238–254. Springer-Verlag LNCS 1145.

Pugh, W. and Wonnacot, D. 1998. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems 20,* 3 (May), 635–678.

Rossie, J. G. and Friedman, D. P. 1995. An algebraic semantics of subobjects. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95).* Austin, TX, 187–199. *ACM SIGPLAN Notices* 30(10).

ROUNTEV, A. AND RYDER, B. G. 2000. Practical points-to analysis for programs built with libraries. Tech. Rep. dcs-tr-410, Rutgers University. February.

SHAPIRO, M. AND HORWITZ, S. 1997. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages*. Paris, France, 1–14.

SIFF, M. AND REPS, T. 1997. Identifying modules via concept analysis. In *Proc. International Conference on Software Maintenance*. Bari, Italy, 170–179.

SNELTING, G. 1996. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology 5,* 2 (April), 146–189.

SNELTING, G. 1998. Concept analysis – a new framework for program understanding. In *Proc. ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. Montreal, Canada, 1–10. *ACM SIGPLAN Notices* 33(7).

SNELTING, G. AND TIP, F. 1998. Reengineering class hierarchies using concept analysis. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Orlando, FL, 99–110.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages*. St. Petersburg, FL, 32–41.

STRECKENBACH, M. AND SNELTING, G. 2000. Points-to analysis for object-oriented languages. Tech. rep., Universität Passau, Fakultät für Informatik. To appear.

SWEENEY, P. F. AND TIP, F. 1998. A study of dead data members in C++ applications. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*. Montreal, Canada, 324–332. *ACM SIGPLAN Notices* 33(6).

SWEENEY, P. F. AND TIP, F. 2000. Extracting library-based object-oriented applications. In *Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE'2000)*. San Diego, CA. To appear.

TIP, F., CHOI, J.-D., FIELD, J., AND RAMALINGAM, G. 1996. Slicing class hierarchies in C++. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*. San Jose, CA, 179–197. *ACM SIGPLAN Notices* 31(10).

TIP, F., LAFFRA, C., SWEENEY, P. F., AND STREETER, D. 1999. Practical experience with an application extractor for Java. In *Proceedings of the Fourteenth Annual Conference on Object-Oriented Programming, Languages, and Applications (OOPSLA'99)*. Vol. 34. 292–305.

TIP, F. AND SWEENEY, P. 1997. Class hierarchy specialization. In *Proceedings of the Twelfth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*. Atlanta, GA, 271–285. *ACM SIGPLAN Notices* 32(10).

TIP, F. AND SWEENEY, P. F. 2000. Class hierarchy specialization. *Acta Informatica*. to appear.

WILLE, R. 1982. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered Sets*, 445–470.

Received December 1999; accepted May 2000