Understanding class hierarchies with KABA

M. Streckenbach, G. Snelting
Universität Passau

Abstract

KABA is a prototype implementation of the Snelting/Tip analysis [2, 3] for JAVA. KABA combines dataflow analysis, type inference and concept lattices in order to perform a fine-grained analysis of member-access patterns in a class hierarchy together with a given set of applications. KABA computes a transformed hierarchy which is guaranteed to be 1. operationally equivalent, 2. maximally factorized, 3. minimal. The new hierarchy in particular makes obvious which classes can be splitted and which cannot; which inheritance relations must be retained and which can be discarded. The paper presents several case studies on medium-sized JAVA programs.

1 Overview

We present the first case studies of a new method for analyzing the usage of a class hierarchy. This method – which was introduced in [2] and is described in more detail in [3] – provides a fine-grained analysis of member access patterns, and enables semantics-preserving reengineering transformations. Our method can analyze a class hierarchy along with any number of programs that use it, and provide the user with a combined view reflecting the usage of the hierarchy by the entire set of programs.

The method provides amazing insight into the de-facto behaviour of a hierarchy. It consists of the following steps:

1. *Points-to analysis* determines for every pointer a conservative (and sharp) approximation of the object set it may point to at runtime; this is used to approximate dynamic binding.
2. *type constraints* essential for program behaviour are extracted from the source text.
3. all *member accesses* (that is, the conservative [and sharp] approximation of their runtime behaviour) are extracted.
4. the type constraints are applied, and finally a *concept lattice* is computed.

The lattice can be interpreted as a new class hierarchy which is guaranteed to be *operationally equivalent* to the old one, but *minimal* and *maximally factorized*. In this workshop paper we will not explain the underlying theory; the interested reader is referred to [1] for a general overview of concept lattices in software technology, and to [2, 3, 5, 6] for the theoretical background of our analysis and its extensions for JAVA and C++. We will however present several case studies we performed with KABA[1], a prototype implementation of the analysis for JAVA. Our examples are JAVA programs of up to 15,000 LOC, and while some turned out to be reasonably structured, KABA revealed high reengineering potential for others.

1.1 A small example

Let us begin with a small example in order to illustrate the method. Figure 1(a) shows a small class hierarchy, in which a class `Person` is defined that contains a person's `name`, `address`, and

[1] KABA = KlassenAnalyse mit BegriffsAnalyse (class analysis via concept analysis). KABA is also a popular chocolate drink in Germany.

```
class String { /* details omitted */ };
class Address { /* details omitted */ };
enum Faculty { Mathematics, ComputerScience };
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, ComputerScience, p2addr);
  p2->hireAssistant(s2);
  return 0;
}
```

(c)

Figure 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).

`socialSecurityNumber`. Classes `Student` and `Professor` are derived from `Person`. `Students` have an identification number (`studentId`), and a thesis `advisor` if they are graduate students. A constructor is provided for initializing `Students`, and a method `setAdvisor` for designating a `Professor` as an advisor. `Professors` 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.

Figure 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.

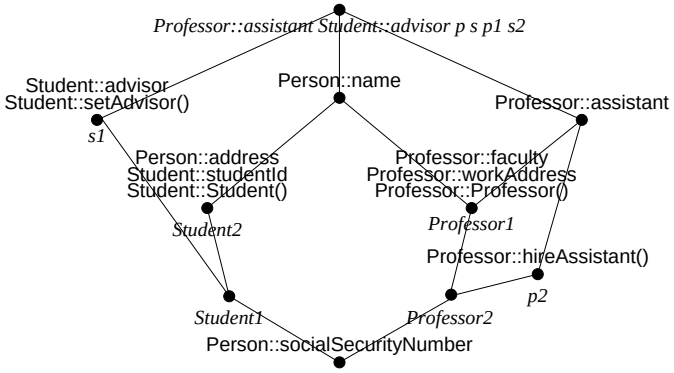

Figure 2: Lattice for Student/Professor example.

- 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[2]:

- Data member `Person::socialSecurityNumber` is never accessed (i.e. dead), 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 data member `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 attribute `Student::advisor` does not appear above attribute `Student::Student()` in the lattice[3].

One can easily imagine how the above information might be used as the basis for restructuring the class hierarchy. First of all, the lattice provides detailed insight into the de-facto behaviour of the hierarchy: due to the principle of conservative approximation, it is guaranteed that a variable will not access a member if it does not appear below the member in the lattice. Furthermore, the lattice is maximally factorized (in particular, common attributes are factored out in superclasses) and minimal (it is the smallest lattice which is operationally equivalent and maximally factorized).

[2]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).

[3]`Student::Student()` also represents the `this`-pointer of the method.

One possibility would thus be a tool to automatically generate restructured source code from the information provided by the lattice, similar to the approach taken in [5, 6].

However, from a redesign perspective, we believe that an interactive approach would be more appropriate in order to improve software-technological criteria such as high cohesion, low coupling, and long-term stability. 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 programmer 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 don't. In an interactive tool, this distinction could be removed by *merging* the concepts for `Professor1` and `Professor2`.

2 Case studies

2.1 jEdit

Our first example is "jEdit", a text editor with useful features like syntax coloring and regular expression search (about 12000 LOC including JavaDoc documentation)[4]. Figure 3 shows the original hierarchy of all classes shipped with "jEdit". Five separate subsystems are visible, concerned with – from top to bottom – 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.

The lattice calculated by KABA (figure 4) consists of several independent substructures, which correspond to the subsystems from the original hierarchy. Most of the original singleton classes, as well as the "input mode" subsystem, exactly reproduce in the right hand part of the lattice, showing no reengineering potential. Note also that three subsystems ("editor commands", "editor modes", "syntax highlighting") are based on dynamic typing: the classes of their objects are computed at runtime. These subsystems therefore do not reappear in figure 4.[5]

One single class however has become a diamond (figure 5). Hence KABA demonstrates that this class can be split up into several classes, because only the objects below the diamond's bottom node need access to all members of the original class.

Most interesting however is the leftmost part of the lattice: the "regular expressions" subsystem has become a complex structure, exposing lots of details about the use of the original classes (figure 6). Some subclasses of the original base class "REToken" are just reproduced by KABA (e.g. "RETokenBackRef", "RETokenEnd" at "RETokenRange" on the right side of figure 6). But the original subclass "RE" has been distributed to many different nodes (left hand side of figure 6); most of them representing non-abstract classes. Thus the lattice presents an optimal splitting and

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

[5] None of the available analysis methods for statically typed languages can handle dynamic typing. Some rely on additional user information, while other reengineeering tools – including KABA – just leave such code unchanged.

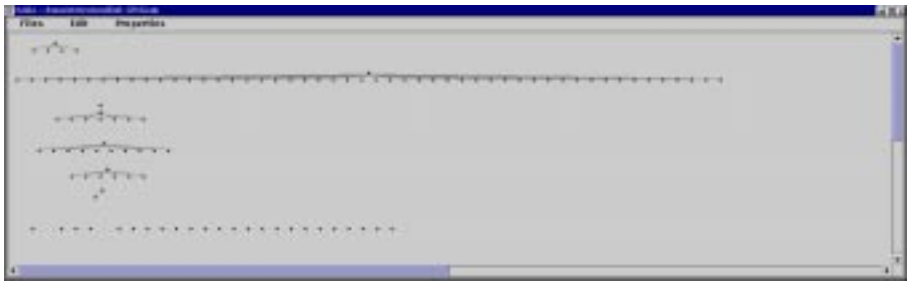

Figure 3: Original class hierarchy for "jEdit" program

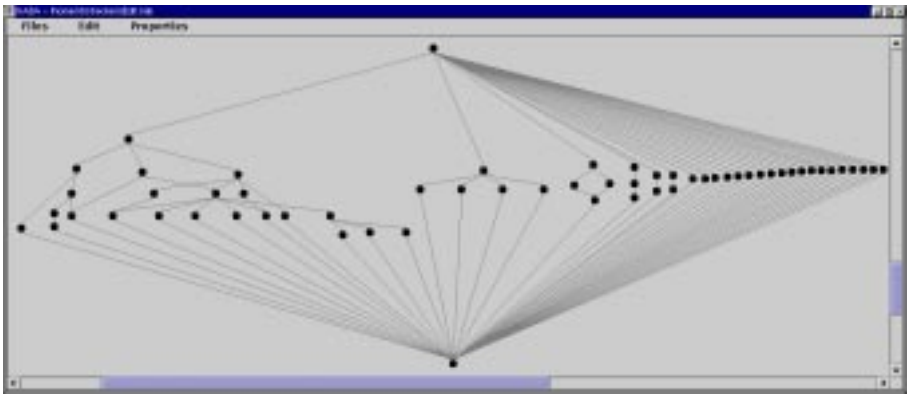

Figure 4: Lattice for "jEdit"

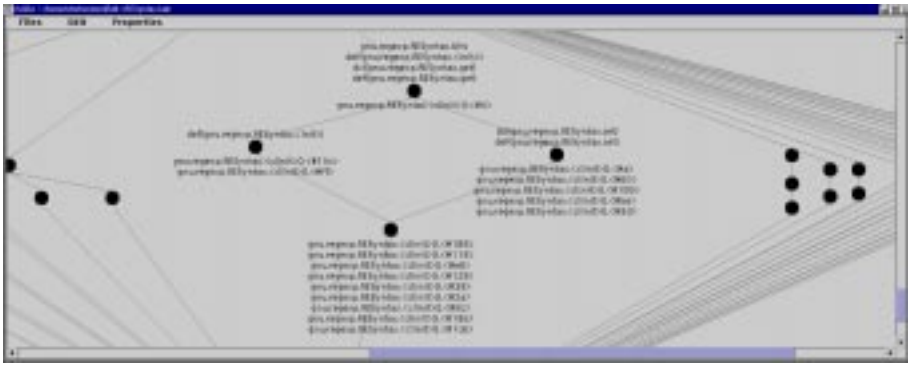

Figure 5: Details for "jEdit": nodes for original "RESyntax" class

factoring for the original subclass "RE", which is guaranteed to be operationally equivalent. Note that the reengineer might merge some of the class proposals again, due to cohesion and coupling considerations. KABA will prohibit merges which might corrupt operational equivalence.

2.2 JAS

The second example is "JAS", a java bytecode assembler, including a Scheme-like scripting language (about 5400 LOC)[6]. Its original class hierarchy is shown in figure 7. Among various single classes and three small inheritance trees it shows a huge structure with more than 50 classes at the top. These classes are part of the scripting language implementation. The top class is "Obj" and all but 4 classes derived from "Obj" additionally implement an interface "Procedure". Each of these classes represents a function like "Add" or "Sub" in the scripting language.

In the KABA lattice (figure 8) this huge structure is reproduced basically unmodified (the figure does not show the whole lattice), demonstrating the original design was good. More interesting in this example is one of the small trees, namely the one with base class "Insn". The leftmost part in the lattice contains "Insn" and its original subclasses (figure 9). All but one subclass of "Insn" contained only a different constructor, and these subclasses are reproduced identically.

The rightmost node however contains all members of the original subclass "Label". This new class is different from the others, because it does not use the methods "Insn.size" and "Insn.write" like all other subclasses. A closer look reveals that these classes are all dealing with the representation of certain bytecode instructions, but "Label" is about bytecode addressing. The implementations of "size" and "write" in "Label" do not contain any code, so they can be considered amputated. An even closer look reveals that the "resolv" function does not execute any useful

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

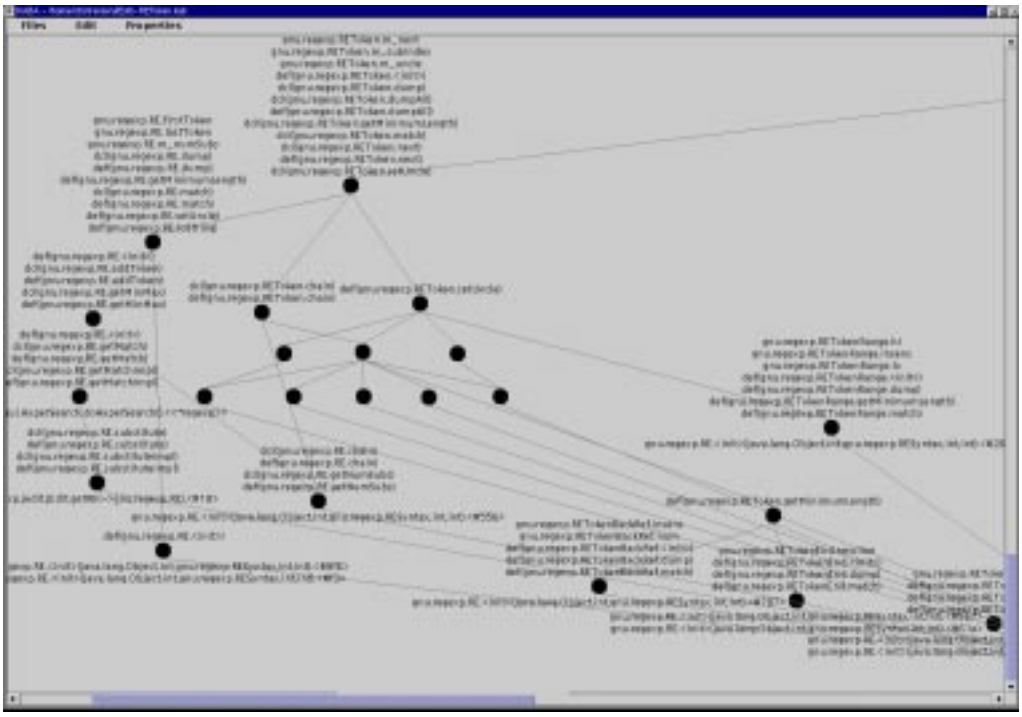

Figure 6: Details for “jEdit”: substructure for “REToken” (not all labels shown)

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 new classes for subclasses of the “InsnOperand” class show a similar phenomenon (figure 10). Two classes (“UnsignedByteWideOperand” and “IincOperand” on the left hand side) are separated from the rest, just like “Label” was. They have own implementations of the method “writePrefix”, while all other subclasses share the same implementation. A look at the source code reveals that this time the other classes use a dummy implementation of “writePrefix” which has no functionality; only the separated classes actually have code for “writePrefix”. This demonstrates “writePrefix” can be removed from “InsnOperand” and put into a new class, from which the separated classes can be derived.

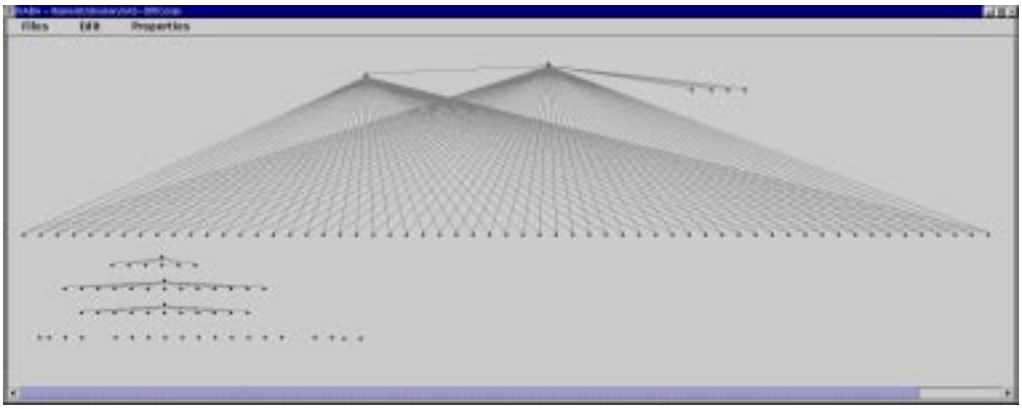

Figure 7: Original class hierarchy for the “JAS” example

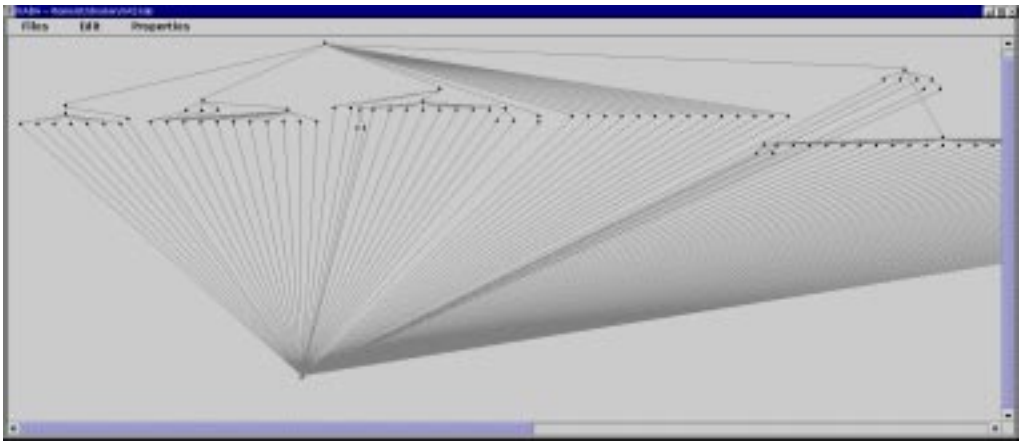

Figure 8: Lattice for "JAS"

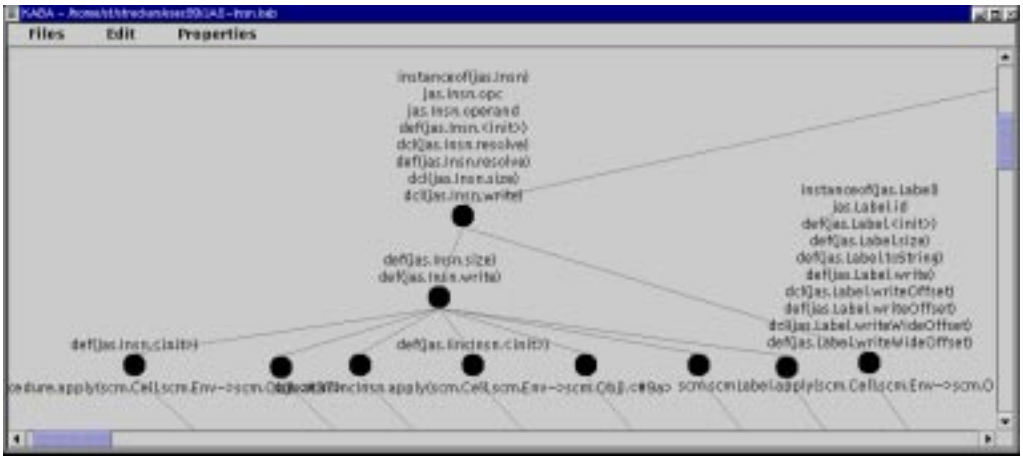

Figure 9: Details for "JAS": substructure for "Insn" (not all labels shown)

2.3 Hanoi

Our last example is a program called "Hanoi". This program is an interactive applet version of the well-known "Towers of hanoi" problem, shipped with *Jax* [4]. The original class hierarchy is shown in figure 11. The lattice derived with KABA is shown in figure 12.

The resulting lattice is comparable in size to the original hierarchy. In fact, the structure of both hierarchies is quite similar, indicating that reengineering potential is quite low. For example, the GUI class with three different subclasses for different platforms in the left part of figure 11 replicates exactly in the middle of figure 12. In some cases however KABA proposes to split classes: figure 12 also presents a detail, where a class from the original program has been split into two subhierarchies; the second of these has again a subclass. A look at the source code reveals that this proposals makes perfect sense from a software engineering viewpoint.

In the right part of the lattice, the little subhierarchy with three subclasses from figure 11 becomes a complex sublattice, namely the rightmost substructure in figure 12. The original classes use two different constructors, which generates two sub-substructures in the lattice. But in order to maintain high cohesion, the original classes should not be splitted. Note that much of the subhierarchy with four subclasses (lower middle in figure 11) is unused and therefore does not reappear in figure 12.

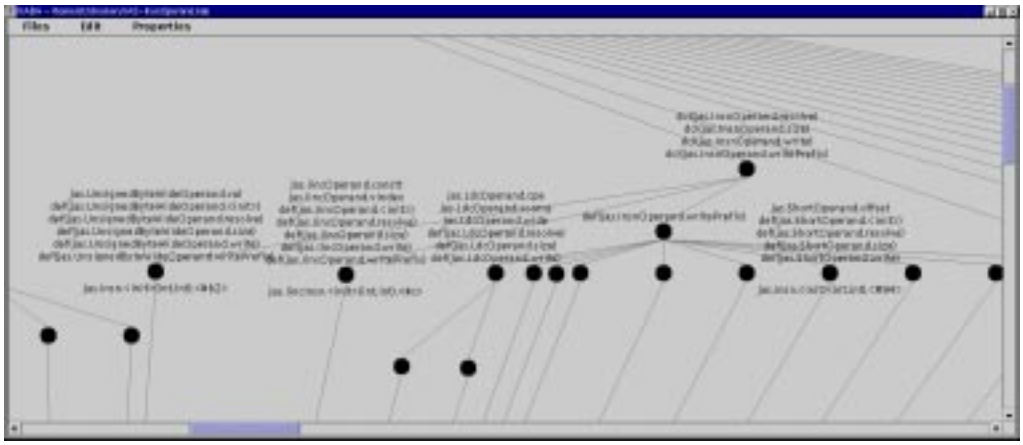

Figure 10: Details for "JAS": substructure for "InsnOperand" (not all labels shown)

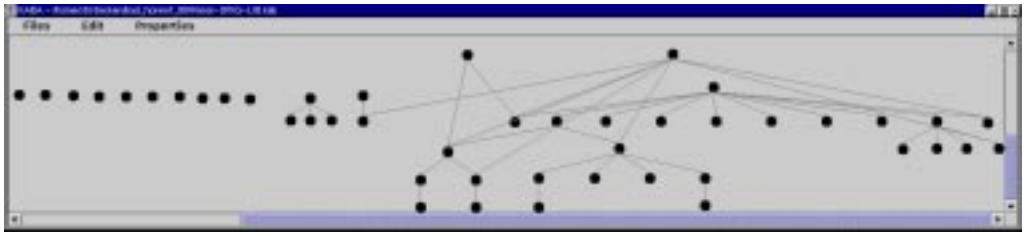

Figure 11: Original class hierarchy for "Hanoi" program

3 Conclusion

Our analysis is perhaps the most expensive analysis of object-oriented programs available at the moment. But is is also one of the most powerful methods, due to its unique combination of points-to analysis, type constraints, and concept lattices. The method includes classic analyses such as dead members or useless variables as special cases. The structure theory of concept lattices (not discussed in this paper) provides lattice simplifications which preserve operational equivalence, but increase quality factors such as low coupling and high cohesion.

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 assesment during initial development. It turned out that the JAVA examples we analysed were all reasonably well structured, and of course

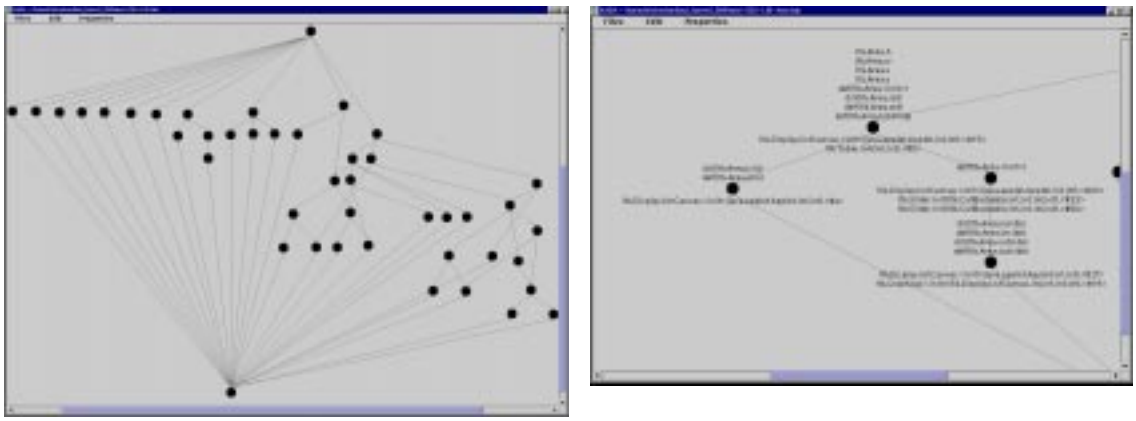

Figure 12: "Hanoi" lattice with details

the real "market" for the method are big old C++ programs. However the complexity of both the language and the method itself seem to prohibit an application to C++ right now. We hope that this situation will change within the next two years.

Acknowledgements. Frank Tip, beeing a co-inventor of the method, provided valuable suggestions for the case studies. Andreas Bögeman supported the implementation of KABA. This work is funded by the Deutsche Forschungsgemeinschaft, grant Sn11/7-1.

References

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

[2] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 99–110, Orlando, FL, November 1998.

[3] G. Snelting and F. Tip. Reengineering of class hierarchies using concept analysis. *Submitted for publication*, 1999.

[4] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Size matters: reducing the size of java class file archives. In *Proc. OOPSLA '99*, 1999. to appear.

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

[6] F. Tip and P. F. Sweeney. Class hierarchy specialization. Technical Report RC21111, IBM T.J. Watson Research Center, February 1998. Submitted for publication.