Concept Lattices in Software Analysis

Gregor Snelting
Universität Passau

Abstract

About ten years ago, the first serious applications of concept lattices in software analysis were published. Today, a wide range of applications of concept lattices in static and dynamic analysis of software artefacts is known. This overview summarizes important papers from the last ten years, and presents three methods in some detail: 1. methods to extract classes and modules from legacy software; 2. the Snelting/Tip algorithm for application-specific, semantics-preserving refactoring of class hierarchies; 3. Ball's method for infering dynamic dominators and control flow regions from program traces. We conclude with some perpectives on further uses of concept lattices in software technology.

1 Overview

Concept lattices were already introduced more than 50 years ago in Birkhoff's first book on lattice theory.[1] More than 20 years ago, Ganter and Wille started to expand the theory considerably and investigated serious applications of concept analysis e.g. in the social sciences. But only 10 years ago, a few researchers started to explore the possibilities of concept lattices for computer science, in particular software technology. Godin, Mili and their coworkers in Montreal applied concept analysis to software design, in particular object-oriented design; this line of reasearch is described in Godin's contribution to the current book. The current author and his group, then in Braunschweig, came up with the first applications of concept lattices in software analysis. Meanwhile, a wealth of results is available, and it is the goal of this article to present important uses of concept lattices and their structure theory for static and dynamic analysis of software artefacts.

It is very natural to apply concept lattices for software analysis, as every software artefact contains an abundance of relations between "objects" and "attributes". To explore hidden structure in such relations is a natural task whenever one wants to understand old software artefacts, or reengineer legacy systems. As a result, a wave of concept lattice applications in software technology was proposed. Some of the applications were well-motivated and based in a thorough understanding of the underlying theory, while others just generated lattices from "yet another relation", without validating the resulting structures. In the following we will concentrate on some substantial contributions; some of the latest papers are covered as well as "classics". Thus in the intention of the author, the current article also serves as a successor to the earlier overview articles [Sne98, Sne00].

[1] We will not give references to general literature on concept analysis or software technology, but restrict ourselfes to citations of specific papers which utilize concept lattices in software analysis.

1.1 Infering configuration structures from source code

One of the very first nontrivial applications of concept lattices for software analysis was Krone's and Snelting's work on the inference of configuration strcutures from source code. The paper was presented in 1994 at the International Conference on Software Engineering (ICSE) [KS94], and an expanded version later appeared in the ACM Transactions on Software Engineering and Methodology [Sne96]. The authors analysed the relationship between code pieces and preprocessor variables in Unix system software, as the preprocessor is typically used for configuration management in older Unix programs. Not only did implications and interferences between configurations become visible in the lattice; the structure theory of concept lattices (irreducible elements and implication base) allowed for a restructuring of the preprocessor variables, and the configuration space could be modularized according to algebraic decompositions of the lattice.

1.2 Identifying modules and classes in legacy software

"Modularization" was also the keyword for a whole series of papers which came out in the following years; triggered by the Y2K problem and its corresponding reengineering challenges. Old legacy systems typically have been developed without modern software technology; in particular, there is no explicit modularization. Identifying modules or classes in legacy code therefore is an important task in order to make such systems survive ("software geriatry" [Parnas]), and concept analysis turned out to be quite helpful.

The author's approach to modularize old Fortran systems, which was presented at ICSE 1997 [LS97], will be described later in detail. Generally speaking, it explores the relationship between program variables and procedures in order to identify modules. At the same time, Siff and Reps presented a similar approach to the restructuring of C programs [SR97]. Van Deursen and Kuipers applied basically the same idea to Cobol legacy programs and published it at ICSE 1999 [vDK99]. All three papers have shown that it is not enough to just compute the lattice, but that background knowledge has to be exploited for a careful selection of "objects" and "attributes", and that the lattice must be simplified, decomposed and interpreted by experts. Other authors have stepped into the footsteps of these three publications, but not with the same success and impact.

1.3 Software component retrieval

A side line of the work on modularization resulted in support for software component retrieval. Godin et al. were probably the first authors to apply concept lattices for component retrieval [GMA93]. Lindig's dissertation [Lin99] went a considerable step further: it carefully engineered concept-based component retrieval and validated its effectiveness for interactive retrieval. Later Fischer combined Lindig's approach with formal specifications, where match relations between specifications are checked beforehand by a theorem prover in order to obtain the initial table from which the lattice is generated; this work won the best Paper Award at the Conference on Automated Software Engineering 1998 [Fis98]. Other authors have proposed similar approaches, but not with the same success and impact.

1.4 Refactoring class hierarchies

As concept lattices are natural inheritance structures, a natural application field is class hierarchies for object-oriented languages. The work by Godin as well as Hesse's approach to requirements engineering aim at the construction of a class hierarchy from some requirements and are described elsewhere in this book. But in practice, evolution of existing systems is more important than the construction of new systems, and in the object-oriented world, refactoring of class hierarchies is the method of choice. Refactoring applies a sequence of (hopefully) semantics-preserving transformations to a class hierarchy such as moving methods to other classes, splitting classes, or extracting new methods from statements. The overall goal is to improve the hierarchy according to software engineering principles such as high cohesion and low coupling, or to identify design patterns in existing code.

One approach was proposed by Tonella, who used concept lattices to identify design pattern in existing code, and reports some success for small examples [TA99]. The semantics-preserving refactoring method by Snelting and Tip was first published at the 1998 Symposium on Foundations of Software Engineering (FSE) [ST98]; a much more detailed version appeared later in the ACM Transactions on Programming Languages and Systems [ST00]. It is based on a fine-grained analysis of a given hierarchy together with a set of applications, and will be explained later in more detail. Work on refactoring using concept lattices is still ongoing, and we will see more results in the future.

1.5 Dynamic analysis

Only a few years ago, the program analysis community started to pay attention not only to static program analysis, but also to dynamic program analysis. Static analysis relies on the source text alone, and precise static analysis is often expensive, if not undecideable. Dynamic analysis uses a set of execution traces in addition; of course the analysis results are valid only for a set of specific inputs or program runs, but are much cheaper to compute and in practice often quite sufficient.

It is therefore not surprising that concept lattices were used for dynamic analysis as well. The first paper presenting such an approach was Ball's reconstruction of control flow graphs from program traces; it was published at FSE 1999 [Bal99] and will be explained later in this article. A more general (but also less precise) approach was presented by Koschke in 2001 and received the Best Paper Award at the International Conference on Software Maintenance [EKS01].

The latest article in this line of research was presented by Ammons, Mandelin, Bodik and Larus at the Conference on Programming Language Design and Implementation (PLDI) [AMBL03]. The authors use concept lattices to debug specifications in temporal logic. The idea is to analyse execution traces (e.g. counterexamples generated by a model checker) and group similar traces into "concepts"; this reduces the debugging work. The paper is not only remarkable due to its high-tech combination of temporal specifications, model checkers, specification extractors, and concept lattices, but also due to the fact that two authors are academics, one is from Microsoft, and one is from IBM.

```
SUBROUTINE R1(...)
COMMON /C1/ V1,V2
...
END

SUBROUTINE R2(...)
COMMON /C2/ V3,V4
COMMON /C3/ V5
...
END

SUBROUTINE R3(...)
COMMON /C2/ V3,V4
COMMON /C4/ V6,V7,V8
...
END

SUBROUTINE R4(...)
COMMON /C2/ V3,V4
COMMON /C3/ V5
COMMON /C4/ V6,V7,V8
...
END
```

	V1	V2	V3	V4	V5	V6	V7	V8
R1	×	×						
R2			×	×	×			
R3			×	×		×	×	×
R4			×	×	×	×	×	×

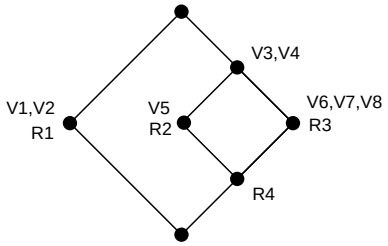


Figure 1: A Fortran fragment, its context table, and its concept lattice

1.6 Impact

These days, citation databases are gaining influence, and we therefore browsed the CiteSeer data base, which contains most publications and citations in computer science. The above overview contains only the most important publications; CiteSeer lists about 40 papers on "concept analysis" or "concept lattices", and more are coming out. The articles sketched above have all been published at very selective and influential conferences and journals (for example, PLDI is according to CiteSeer the most cited of all computer science conferences), and as a consequence CiteSeer lists more than 400 citations of concept analysis papers in computer science. Ganter and Wille made it into CiteSeer's list of the 10000 most cited computer scientists, even though they are mathematicians.

This success would not have been possible without readily available implementations. In Germany, the concept lattice software from Wille's group is quite well-known, but on an international scale the most popular software is Lindig's implementation of Ganter's algorithm for concept lattice generation [Lin], as it is very efficient, robust, and usable as a background tool without own GUI. The software has been installed at ca. 50 sites worldwide.

2 Modularization

Let us now describe our work on modularization of old Fortran programs in some detail. While the above-mentioned later papers by Siff/Reps and vanDeursen/Kuipers were more successful from a practical viewpoint, our work introduced the basic idea. The project was based on a cooperation with a national research institution, who aimed at reengineering their aerodynamics software written in Fortran. Several approaches to modularize the system had failed, so it was decided to try concept lattices.

The fundamental idea is to investigate the relation between global variables and procedures. If a set of variables V and a set of procedures P can be identified, where all procedures in P use only variables in V, and all variables in V are only used by procedures in P, then P together with V is definitely a module candidate. The reason is that modules implement information hiding, hence a module's variables may only be accessed

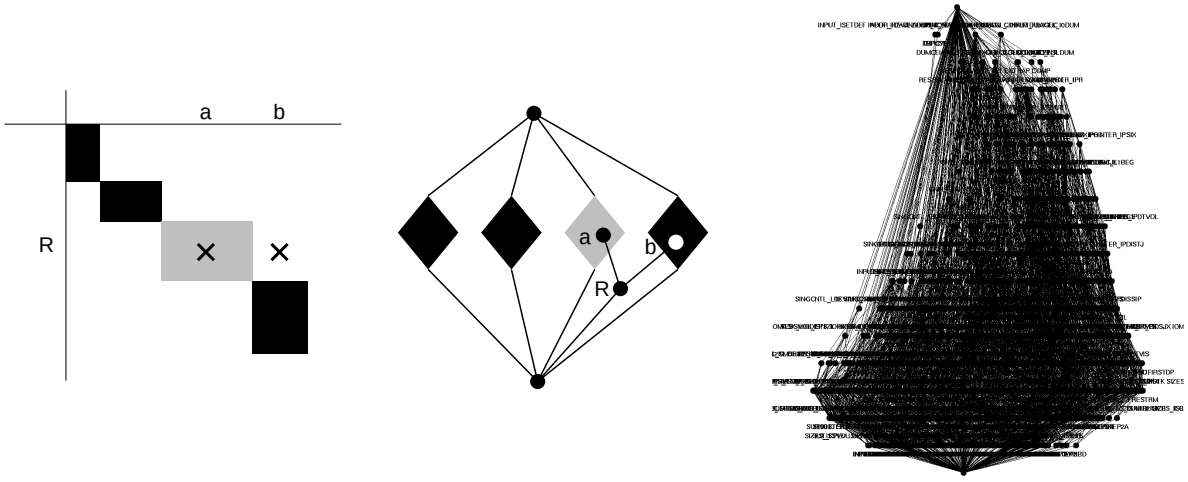


Figure 2: Horizontally decomposable lattice and an interference (left); Concept lattice for Fortran aerodynamics program (right)

through its interface procedures. Figure 1 presents a small example of four procedures, acting on various global variables which are organized in several “Common” blocks. The goal was to identify modules as described, and restructure the “Common” blocks such that there is one “Common” block per module. Figure 1 also presents the context table extracted from the source code, and the corresponding concept lattice.

The general situation is depicted in figure 2 (left part). Modules correspond to rectangle *shapes* in the context table, but must not be completely filled rectangles, as not every procedure accesses all module variables (remember that in the context table, row and column permutations do not matter!). The corrresponding lattice is horizontally decomposable, and every rectangle shape in the table corresponds to one horizontal summand. Figures 1 and 2 both present horizontally decomposable lattices, hence a modularization is possible. In case horizontal summands are connected by a few additional infima, these are called interferences. Interferences prevent modularization (as the information hiding principle is violated), but can usually be removed by some small behavior-preserving transformations of the source code.

Now let us come back to our project with the national research institution. We analysed a 106 KLOC Fortran program, which was 25 years old and had undergone countless modifications. 317 procedures were acting on 492 global variables, distributed over 40 “Common” blocks. After extraction of the context table, the lattice was computed and layoutet. The result can be seen in figure 2 (right). The lattice has more than 2000 elements, is definitely not decomposable, but consists basically of interferences. A modularization based on a repartitioning of the global variables is therefore not possible. The national institution decided to cancel the reengineering project and develop a new system.

Let us add that the basic method can be extended in various ways: Siff/Reps not only used variables and procedures, but also types, and they explicitely coded the fact that $p \in P$ does *not* use variable $v \in V$ or type t. vanDeursen/Kuipers preprocessed the variables, in order to distingish temporary variables from those relevant to modules.

```
class A {
  int x, y, z;
  void f() {
    y = x;
  }
}

class B extends A {
  void f() {
    y++;
  }
  void g() {
    x++;
    f();
  }
  void h() {
    f();
    x--;
  }
}
```

```
class Client {
   public static void
   main(String[] args) {
     A a1 = new A();  // A1
     A a2 = new A();  // A2
     B b1 = new B();  // B1
     B b2 = new B();  // B2

     a1.x = 17;
     a2.x = 42;
     if (...) { a2 = b2; }
     a2.f();
     b1.g();
     b2.h();
   }
}
```

	A.x	A.y	A.z	dcl(A.f)	def(A.f)	dcl(B.f)	def(B.f)	dcl(B.g)	def(B.g)	dcl(B.h)	def(B.h)
a1	×										
a2	×			×							
b1								×			
b2						×				×	
A1											
A2					×						
B1							×		×		
B2							×				×
A.f.this	×	×			×						
B.f.this		×					×				
B.g.this	×					×			×		
B.h.this	×					×					×

Figure 3: A small example program and its member access table

For the Fortran analysis, Wegman proposed to transform the program into static single assignment form first, as Fortran programs often misuse the same variable for different purposes. All this will improve the results of modularization. But today, with a few years distance, the author does not really believe in automatic modularization any more, because really old programs are just too chaotic. Even concept lattices will not prevent their entropy death.

3 Automatic refactoring

The Snelting/Tip algorithm is one of the most complex, but also most powerful applications of concept lattices. It serves to automatically restructure ("refactor") a given class hierarchy with respect to a given set of client programs. As clients typically do not access every feature of a given hierarchy, the result is a refactored hierarchy which is "specialised" or "taylored" to the specific clients. In particular, all objects will contain only members and methods they really need (with respect to client behaviour). The method combines program analysis, type constraints and concept analysis to compute the most fine-grained refactoring which is still preserving client behaviour.

In this section, we recapitulate the basic properties of this algorithm. Full details can be found in [ST00].

3.1 Collecting member accesses

The algorithm is based on a fine-grained analysis of object access patterns. For all objects or object references o, it determines whether o does access member m from class C. The result is a binary relation, coded in form of a table T.

As an example, consider the program fragment in figure 3 (left). B, being a subclass

of A, redefines $f()$ and accesses the inherited fields x, y. The main program creates two objects of type A and two objects of type B, and performs some field accesses and method calls. Table T for this example contains of rows labelled with object references $a1, a2, b1, b2, A.f.this, B.f.this$, $B.g.this, B.h.this$ as well as object creation sites $A1, A2, B1$, $B2$. Columns are labelled with fields and methods $A.x, A.y$, $A.z, A.f(), B.f()$, $B.g(), B.h()$. For methods, there is an additional distinction between declarations and definitions ($dcl(C.f())$ vs. $def(C.f())$), which makes the analysis much more precise.

Points-to analysis is used to determine for an object reference o to which object creation sites it might point to at runtime; this set is denoted $pt(o) = \{C1, C2, \ldots\}$. $pt(o)$ may be too big (i.e. unprecise), but never too small (i.e. pt is a conservative approximation). Now let $Type(o) = C$ be the static type of o, and let member accesses $o.m$ resp. $o.f()$ be given. Table T will contain entries $(o, C.m)$ resp. $(o, dcl(C.f()))$. Furthermore, entries $(O, def(C.f()))$ are added for all $O \in pt(o)$ where $C = StaticLookup(Type(O), f)$. For the above example, the resulting table is shown in figure 3 (right).

3.2 Type constraints

In a second step, a set of type constraints is extracted from the program, which are necessary for preservation of behaviour. The refactoring algorithm computes a new type (i.e. class) for every variable or class-typed member field, and a new "home" class for every member. Therefore, constraints for a variable or field x are expressed over the (to be determined) new type of x in the refactored hierarchy, $type(x)$; constraints for a member or method $C.m$ are expressed over its (to be determined) new "home class", $def(C.m)$.

There are basically two kinds of type constraints:

1. Any (explicit or implicit) assignment `x = y;` in the program text gives rise to a type constraint $type(y) \leq type(x)$. Such constraints are called assignment constraints.

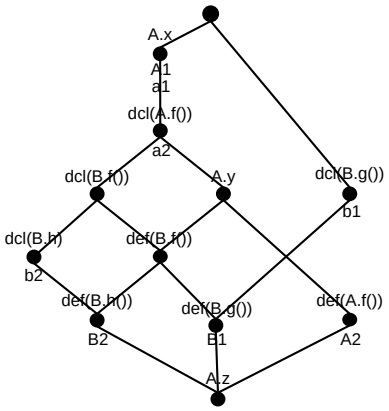
2. If subclass B of A redefines a member or method m, and some object x accesses both $A.m$ and $B.m$ (that is, $\exists x : (x, def(A.m)) \in T \wedge (x, def(B.m)) \in T$), then $def(B.m) < def(A.m)$ must be retained in order to avoid ambiguous access to m from x. Such constraints are called dominance constraints. A more obvious, similar dominance constraint requires that for all methods $C.f$, $def(C.f) \leq dcl(C.f)$.

Once all type constraints have been extracted, they are incorporated into table T. To achieve this, we exploit the fact that a constraint can be seen as an *implication* between table rows resp. columns, and that there is an algorithm to incorporate any given set of implications into a table. First we observe that even in the refactored hierarchy, a subtype inherits all members from its supertype. Therefore $type(y) \leq type(x)$ emforces that any table entry for x must also be present for y; that is $\forall m : (x, m) \in T \Rightarrow (y, m) \in T$, or $x \rightarrow y$ for short. Second, $def(B.m) < def(A.m)$ enforces that any table entry for $def(B.m)$ must also be present for $def(A.m)$, which is written as $def(B.m) \rightarrow def(A.m)$.

Reconsidering figure 3, the following assignment constraints are collected in form of implications:

$$A.y \rightarrow A.x, A.f.this \rightarrow a2, B.f.this \rightarrow a2, B.g.this \rightarrow b1,$$
$$B.h.this \rightarrow b2, a1 \rightarrow A1, a2 \rightarrow A2, b1 \rightarrow B1, b2 \rightarrow B2, a2 \rightarrow b2$$

	A.x	A.y	A.z	dcl(A.f)	def(A.f)	dcl(B.f)	def(B.f)	dcl(B.g)	def(B.g)	dcl(B.h)	def(B.h)
a1	×										
a2	×			×							
b1								×			
b2	×			×		×				×	
A1	×										
A2	×	×		×	×						
B1	×	×		×		×	×	×	×		
B2	×	×		×		×	×			×	×
A.f.this	×	×		×	×						
B.f.this		×		×		×	×				
B.g.this	×	×		×		×	×	×	×		
B.h.this	×	×		×		×	×			×	×



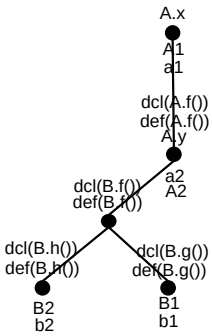


Figure 4: Table after incorporating type constraints for figure 3 (left); corresponding concept lattice and its simplified version (right)

Furthermore, the following obvious dominance constraints are collected:

$$def(A.f) \rightarrow dcl(A.f), def(B.f) \rightarrow dcl(B.f), def(B.g) \rightarrow dcl(B.g), def(B.h) \rightarrow dcl(B.h)$$

as well as the non-obvious dominance constraints

$$def(B.f) \rightarrow def(A.f), dcl(B.f) \rightarrow dcl(A.f)$$

These implications are incorporated into the initial table (figure 3 right) by copying row entries from row y to row x resp. column entries from column $def(A.f)$ to column $def(B.f)$ etc. Note that in general there may be cyclic and mutual dependences between row and/or column implications, thus a fixpoint iteration is required to incorporate all constraints into the table. The final table for figure 3 is presented in figure 4 (left).

3.3 The refactored hierarchy as a concept lattice

In a final step, concept analysis is used to construct the refactored hierarchy from the final table. Concept lattices can naturally be interpreted as inheritance hierarchies. The concept lattice for figure 3, as constructed from the final table (figure 4 left), is given in figure 4 (middle). Every lattice *element* represents a *class* in the refactored hierarchy. Method or field names *above* an element represent the *members* of this class. Objects or pointers *below* an element will have that element (i.e. class) as its new *type*. In particular, all objects now have a new type which contains only the members the object really accesses.

Typically, original classes are split and new subclasses are introduced. This is particularly true for figure 4 (middle), where the raw lattice introduces 12 refactored classes instead of the original two classes. These new classes represent object behaviour patterns: $a1$ and $A1$ use $A.x$ but nothing else, which is clearly visible in the lattice. $a2$ additionally calls $a.f()$ and therefore needs the declaration of this method. $b1$ calls $B.g()$ and nothing else; $b2$ calls $B.h(), B.f()$ plus anything called by $a2$. The "real objects" $A2, B2, B1$ are located far down in the hierarchy and use various subsets of the original hierarchies' members. $B2$ in particular not only accesses everything accessed by $b2$, but also calls $B.f()$, which leads to a multiple inheritance in the lattice. Note that the raw lattice clearly

distinguishes between a class and its interface: several new classes contain only $dcl(...)$ entries, but no $def(...)$ entries or fields, meaning that they are interfaces.

As the lattice respects not only the member accesses, but also the type constraints, it guarantees preservation of behaviour for all clients. The lattice is rather fine-grained, and in its raw form represents the most fine-grained refactoring which respects the behaviour of all clients. But from a software engineering viewpoint, the lattice must be simplified in order to be useful. Some simplifications are quite obvious: "empty" elements (i.e. new classes without own members) such as the top and bottom element in figure 4 (middle) can be removed; multiple inheritance can in many cases be eliminated, and lattice elements can be merged according to certain (behaviour-preserving) rules. In particular, the distinction between a class and its interface can be removed by merging lattice elements. The final result is in general not a lattice anymore, only a partial order – but for object-oriented programming, this is fine.

Figure 4 (right) presents a simplified version of figure 4 (middle), which can be generated automatically. Now the empty elements and the interfaces are gone, and the different access patterns for the objects are visible even better:

- The two objects of original type B have different behaviour, as one calls g and the other calls h. Therefore, the original B class is split into two unrelated classes.
- The two objects of original type A have related behaviour, as $A2$ accesses everything accessed by $A1$, plus $A.f()$. Therefore, the original A class is split into a class and a subclass.
- $A1$ does only contain $A.x$ and not $A.y$. $A.z$ is dead, as it appears at the bottom element in the lattice. Thus objects become smaller in general, as unused members are physically absent in objects of the new hierarchy.

One might think of simplifying even further by merging the two topmost elements in figure 4 (right), but that would make $A1$ bigger than necessary by including $A.y$ as a member. It is the refactorer's decision whether this disadvantage is outweighted by a simpler structure of the refactored hierarchy. If so, the refactoring editor must guarantee that behaviour of all clients is still preserved after simplification.

3.4 The KABA system

KABA (KlassenAnalyse mit BegriffsAnalyse) is an implementation of the Snelting/Tip method for Java. KABA consists of four parts: a static analysis, a dynamic analysis, a graphical class hierarchy editor and a bytecode transformation tool.

KABA will display the (original or simplified) lattice, and offers browsing as well as back links to the original hierarchy. But the true value of the KABA hierarchy editor lies in its ability to manipulate the (refactored or original) hierarchy – where of course preservation of behaviour is always guaranteed. For example, classes can be merged or methods can be moved to neighbour classes. Eventually, Java code can again be generated. Note that all original statements remain unchanged – only the hierarchy and the declarations of variables, fields and methods change, as the classes from the new hierarchy have to used as computed.

Figure 5 shows a KABA screenshot. The reader should be aware that the implementation of the Snelting/Tip method for KABA and its application to real Java programs

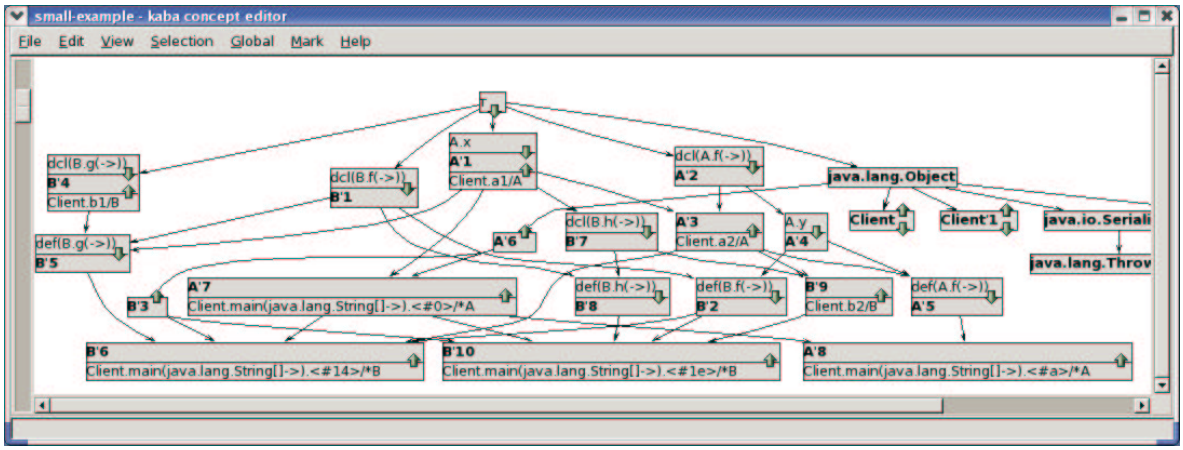


Figure 5: Kaba screenshot for figure 3

is much more complex as described above: the full Java language must be handled as well as libraries; questions of scale-up do matter, etc. Some case studies using KABA on real-world programs can be found in [SS03].

4 Dynamic Analysis

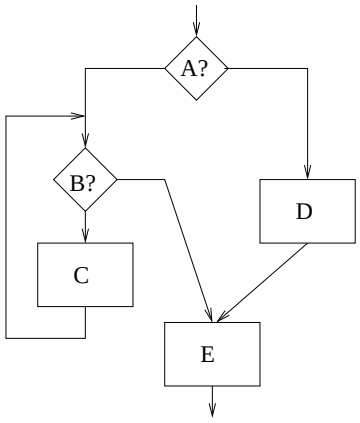
Ball [Bal99] was the first one to use concept lattices for dynamic analysis. His scenario assumes that no source code is given, just an executable program, and the task is to reconstruct the control flow graph (CFG) and its dominator relations. The starting point is an execution profile which for every test run says which statements or functions have been executed.

In order to understand Ball's method more fully, we will have to introduce a few definitions. CFG's are well known, and figure 6 (left) presents a small example. In CFG's, the definition of a dominator is very important. Statement x is a *predominator* of statement y, if x must always be executed before y: every path from the CFG start to y must pass through x. In the standard example, x is a while loop entry point, and y is a statement in the loop body; obviously y can never be executed unless x has been executed. Statement y is a *postdominator* of x if y must always be executed after x: every CFG path from x to the CFG exit must pass through y.

Figure 6 (right) presents a few examples for these definitions. Note that e.g. B is not a predominator for E since E can be reached via D, but B is a predominator for C as there is no way to reach C except via B. The relations x *predom* y and y *postdom* x are partial orders, and in fact pre- resp. postdominators can always be arranged in a tree.

If x *predom* y and y *postdom* x, then x and y are said to be in the same control flow *region*. Statements in the same region are always executed together or not at all; it is easy to see that these regions form an equivalence relation on the CFG nodes. In the example, A and E are in the same region as A *predom* E and E *postdom* A, but other non-trivial regions do not exist (B, C, D are all in their own singleton region).

As defined above, dominators are static relations: they are valid for every possible execution. If source code is missing, static dominators cannot be determined. All that



$$A \textit{ predom } C,\ B \textit{ predom } C$$
$$\neg(B \textit{ predom } D),\ \neg(B \textit{ predom } E)$$
$$E \textit{ postdom } C,\ E \textit{ postdom } D$$

Figure 6: A simple control flow graph and some pre- and postdominator relationships

	add	rotate	rem	Min	Succ	DelFix
t1	X		X			X
t2	X	X	X			X
t3	X	X	X	X		
t4	X	X	X	X	X	X
t5	X	X	X	X	X	X

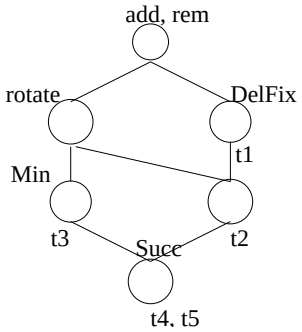


Figure 7: A trace table and its concept lattice

can be said is that for the given test runs, x was always executed before (or after) y. Such *dynamic* dominators are valid only for some specific set of executions, but not for all program executions. The more executions are run, the more likely it is that a dynamic dominator is in fact a static dominator. Thus dynamic dominators are good candidates for static dominators and perhaps allow a reconstruction of the CFG! This was Ball's idea, together with his insight that dynamic dominators can be determined by computing concept lattices from program traces.

Let us consider an example. In figure 7 we see a table summarizing the results of profiling 5 test runs. For every test run (i.e. row) we see which functions among 6 functions were executed. The concept lattice for this table is shown in the same figure. What is the interpretation of this lattice? First of all, the concepts are *dynamic regions*: all functions in a concept's intent are executed together or not at all. Furthermore, upward arcs are implications: any test that executes `min` and `succ` also executes `rotate`. Therefore, `rotate` is a dynamic dominator of both `min` and `succ`! (Unfortunately, we cannot tell whether it is a pre- or postdominator, as the trace table does not say anything about the *temporal order* of function executions.)

Next, suprema resp. infima correspond to forks of control flow ("if-statements"): `add` dominates both `Min` and `DelFix`, but there are tests which distinguish execution of both. Let us assume that `add` is in fact a dynamic *pre*dominator of `min` and `succ`, then there must be a case distinction at `add` leading to either `min` or `succ` (and the case distinction

cannot be earlier in the execution, since `add` is at the supremum!). In this situation, the infimum of `Min` and `DelFix` correponds to the "join point" (dynamic postdominator) in the CFG, where the two branches of the "if" merge again.

But note that the situation could be the other way round, that is the infimum could correspond to the "if" predominating the two branches, and the supremum could be the join point[2]. In any case, the lattice is an order-preserving image of the CFG according to the following equations:

$$x \textit{ predom } y \implies \mu(x) \geq \mu(y)$$
$$x \textit{ postdom } y \implies \mu(x) \leq \mu(y)$$

If the trace table can be enriched with information saying which function was executed earlier, the lattice can definitely distinguish (dynamic) pre- and postdominators. The more test cases are used, the more fine-grained this lattice will become, and in the limit case of an infinite number of tests covering all CFG paths, the CFG can be order-embedded into the lattice. Note that the CFG is only a quasi-order as it usually contains cycles.

Summarizing this preliminary discussion, we see that the concept lattice allows to uncover the control flow graph and its regions and dominators from test cases. This is a very useful method for reengineering old executables where the source code has been lost – a situation which occurs in practice. Let us hope that Ball will proceed to work out the details and apply it to real-world examples.

5 Conclusion

This overview article centered around applications of concept lattices in software analysis. Several other applications of concept analysis in software technology are described elsewhere and have been left out due to space restrictions. Examining the applications we have discussed, one can clearly distinguish two different "historical" phases: early applications of concept lattices in software technology centered on design and static analysis, while later applications are based on program transformation and dynamic analysis.

It is kind of surprising that all these applications stick to the basic theory of concept lattices and their corrresponding implication base, but do not apply more advanced results, such as the structure theory of concept lattices or fuzzy contexts. In fact the author believed for a while that these advanced techniques can improve applications in software technology. But today we know that this is not true. The reason is that realistic lattices do not have the properties required for the advanced techniques. For examples, typical lattices in software technology have neither congruences nor block relations ("weak congruences"); the reason is that congruences have nonlocal effects on the lattice which have no counterpart in the world of software. Similarly, subdirect or subtensorial decompositions could not be found in our various applications.

Nevertheless, concept lattices have received a huge wave of attention by software technology researchers in the last seven years, and proved to be a very helpful instrument. We will see many more concept lattices in software technology in the next seven years!

[2]Ball for some reason assumed that suprema always correspond to predominators, but Ganter pointed out that the dual situation could also be the case.

References

[AMBL03] Glenn Ammons, David Mandelin, Rastislav Bodik, and James Larus. Debugging temporal specifications with concept analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–193, 2003.

[Bal99] Thomas Ball. The concept of dynamic analysis. In *ESEC / SIGSOFT FSE*, pages 216–234, 1999.

[EKS01] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Feature-driven program understanding using concept analysis of execution trace. In *Proc. Ninth International Workshop on Program Comprehension (IWPC'01)*, May 2001.

[Fis98] B. Fischer. Specification-based browsing of software component libraries. In *Automated Software Engineering*, pages 74–83, 1998.

[GMA93] R. Godin, R. Missaoui, and A. April. Experimental comparison of navigation in a galois lattice with conventional information retrieval methods. *International Journal of Man-Machine Studies*, 38, 1993.

[KS94] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proceedings of the 16th international conference on Software engineering*, pages 49–57. IEEE Computer Society Press, 1994.

[Lin] Christian Lindig. Concepts: a program for concept lattices. `http://www.st.cs.uni-sb.de/~lindig/src/concepts.html`.

[Lin99] Christian Lindig. *Algorithmen zur Begriffsanalyse und ihre Anwendung bei Softwarebibliotheken*. PhD thesis, Technische Universität Braunschweig, 1999.

[LS97] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359. ACM Press, 1997.

[Sne96] Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):146–189, 1996.

[Sne98] Gregor 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, 1998. Invited contribution.

[Sne00] Gregor Snelting. Software reengineering based on concept lattices. In *Proc. 4th European Conference on Software Maintenance and Reengineeering*, pages 3–12, 2000. Invited contribution.

[SR97] Michael Siff and Thomas Reps. Identifying modules via concept analysis. In *Proc. International Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.

[SS03] Mirko Streckenbach and Gregor Snelting. Behaviour-preserving refactoring with KABA. In *Submitted for publication*, August 2003.

[ST98] Gregor Snelting and Frank 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.

[ST00] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, pages 540–582, May 2000.

[TA99] Paolo Tonella and Giuliano Antoniol. Object-oriented design pattern inference. In *International Conference on Software Maintenance*, pages 230–, 1999.

[vDK99] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st international conference on Software engineering*, pages 246–255. IEEE Computer Society Press, 1999.