

**Prüfer:** Prof. Dr. Erhard Plödereeder

**Betreuer:** Dr. Rainer Koschke

**Beginn am:** 01.05.2000

**Beendet am:** 31.10.2000

**CR-Klassifikation:** D.2.7, I.1.2, K.6.2

Diplomarbeit Nr. 1855

## **Komponentenerkennung durch Begriffsanalyse**

Alexander Pormann

Institut für Informatik  
Universität Stuttgart  
Breitwiesenstr. 20 - 22  
D-70565 Stuttgart



## Abstract

One aspect of reverse engineering is to identify components of software systems. Some of the more recent approaches to component recovery are based on a mathematical technique called concept analysis. This thesis evaluates four approaches to component recovery using concept analysis. The approaches were implemented and evaluated using benchmarks for component recovery techniques. Quantitative data for recall and precision of these techniques are compared to other more traditional component recovery techniques. Omissions in the description of these techniques in the original papers had to be resolved.

## Acknowledgment

I would like to thank Rainer Koschke who nearly always had time when I needed him. He even took some work home.

I would like to thank the assistants of the department of Programmiersprachen und Compilerbau who helped me, even if they had nothing to do with my work. The atmosphere was great.

---

# Table of Contents

<b>Chapter 1</b>	<b>Introduction.....</b>	<b>1</b>
	1.1 Task Formulation.....	3
	1.2 Overview .....	3
<b>Chapter 2</b>	<b>Bauhaus and Tools .....</b>	<b>5</b>
	2.1 Bauhaus .....	5
	2.2 Rigi.....	7
	2.3 concept .....	8
<b>Chapter 3</b>	<b>Finding Components via Concept Analysis.....</b>	<b>11</b>
	3.1 Mathematical Background .....	12
	3.2 Lindig and Snelting .....	15
	3.2.1 Block Relations .....	17
	3.2.2 Horizontal Decomposition .....	19
	3.3 Sahraoui et al.....	21
	3.3.1 DO Identification.....	22
	3.3.2 Method Identification.....	23
	3.4 Siff and Reps .....	24
	3.5 Graudejus .....	28
	3.6 Combination with other Techniques .....	29
<b>Chapter 4</b>	<b>Design.....</b>	<b>31</b>
	4.1 Overview .....	31
	4.2 Bauhaus structure .....	32
	4.3 The Lattice Building Process .....	33
	4.4 Design of the Concept Analysis Components.....	33
	4.4.1 generic_concept_analysis.....	34
	4.4.2 concepts.....	37
	4.4.3 build_lattice.....	38
	4.5 Design of the Algorithms .....	40
	4.6 Incremental Approach.....	42
<b>Chapter 5</b>	<b>Implementation .....</b>	<b>45</b>
	5.1 Reuse.....	45
	5.2 Lattice Building.....	45
	5.3 Algorithms.....	46
	5.3.1 Lindig and Snelting.....	46
	5.3.1.1 Collapse Phase.....	46

---

---

5.3.1.2 Interference Resolution.....	47
5.3.2 Sahraoui et al.....	48
5.3.2.1 Collection of DO Candidates.....	48
5.3.2.2 Overlap Resolution .....	48
5.3.2.3 Method Assignment.....	49
5.3.3 Siff and Reps.....	49
5.3.4 Graudejus .....	49
<b>Chapter 6    Benchmarks .....</b>	<b>51</b>
6.1 Bauhaus Benchmarks .....	51
6.2 Terminology of the Benchmark.....	52
6.3 Evaluation.....	54
6.3.1 Recall Rate .....	54
6.3.2 Measured Data .....	55
6.3.3 Conclusions .....	56
<b>Chapter 7    Summary.....</b>	<b>59</b>
7.1 Conclusions .....	59
7.2 Future Research.....	59
<b>Appendix A    Bibliography .....</b>	<b>61</b>

---

## *Figure Index*

Reduced Entity–Relationship Diagram for RFGs.....	6
Small Example for a concept Input File.....	8
Result of Program Call.....	8
Result of Program Call.....	9
Example for a Relation Table.....	12
Concepts for Figure 3-1.....	13
Lattice for Figure 3-1.....	14
Sparse Representation of Figure 3-1.....	15
Example for a Variable Usage Table.....	17
Example for a Maximal Rectangle.....	17
Small Example for Cohesion.....	19
A Simple Interference.....	20
Reduced Hierarchy of the Package rfgs.....	32
Building a Concept Lattice.....	34
Call of concept and File Handling.....	35
File lattice.txt: Concept Node Information.....	36
File lattice.gv: Information on the Subconcept Relation.....	37
concepts.ads.....	38
build_lattice.ads.....	39
generic_ca.ads.....	40
components used by the algorithms.....	41
the Perform-function of an analysis.....	42
Example Relation Table.....	43
Example Relation Table with Merged Node.....	43
The Call of concept.....	46
Collapsing Horizontal Summands.....	47
A Single Interference Resolution.....	48
Copy of DO Candidates.....	49
Copy of DO Candidates.....	49
Internal Access Check.....	50
Overlap Resolution Loop.....	50
Suite of Analyzed C Systems.....	52
Number of Atomic Components in Analyzed Systems.....	52
Recall Rate for ADTs.....	55
Recall Rate for ADOs.....	56
ADT Comparison.....	57
ADO Comparison.....	58

---

---

- Reverse Engineering
- Component Detection
- ADT, ADO

---

Maintenance is the dominating phase in the lifetime of a software system. A survey of 487 data processing installations revealed that in 1978 48,8% of the effort was spent on maintenance [Boe81, p 18]. During this long period of maintenance, the systems are enhanced, restructured, and various details or larger parts of the system are changed. In most of the cases, this leads to redundancy and increasing size making the system harder to maintain.

In order to maintain a software system, maintenance engineers have to understand the system, its structure and behavior. Often components of software systems are not adequately documented. This results in an expensive process to identify and document components of the software system. Thus, an automatic process to identify components would be very helpful to maintenance engineers. Some approaches have been proposed ([Yeh95], [LiuWil90], [Can96],...) but a simple automatic solution to component detection seems to be not yet in sight. The quality of the automatically detected components is not good enough to be accepted without additional examination by the maintenance engineers [Kos00]. Thus, most recently proposed approaches to component detection are semi-automatic. The knowledge of the maintenance engineer is supported by automatic component proposals.

One of the goals of the Bauhaus project at the department *Programmiersprachen und Compiler, Institut für Informatik, Universität Stuttgart*, is to investigate various algorithms for identifying components. The results of these algorithms are evaluated using the Bauhaus benchmarks. A maintenance engineer using automatic component recovery techniques should be provided with the knowledge which technique is best for which situation. With this knowledge he should be able to identify module candidates faster and with a higher quality.

This thesis is based on recent approaches on identifying component candidates by a mathematical method called “concept analysis”. Four of these approaches

have been implemented and evaluated with existent systems in this thesis. The results are presented and compared with other more traditional approaches.

A *component* is a set of related entities. An *atomic component* (also called logical module) consists only of base entities. *Base entities* are subprograms, global variables, and user-defined types. A *subsystem* may be hierarchical, i.e., may contain other components. A *component candidate* is a component proposed by an automatic technique. The component candidates detected by the approaches are ADTs (abstract data types) and ADOs (abstract data objects). Rainer Koschke describes ADTs and ADOs as follows [Kos00, p 87f]:

*“At a higher level of abstraction, an abstract data type consists of a domain of values for the type and some allowed operations on that type. In an implementation of an abstract data type, the domain of values is implemented by a data structure which is read and set by routines – its operations. The user of an abstract data type can declare objects of that type and pass them as actual parameters to the operations. Consequently, it is a necessary prerequisite for operations of an abstract data type to mention the data type in their signature, i.e., their parameter list or their return type in the case of functions. That is, all routines with a data type T in their signature are candidates for an operation of the abstract data type T. However, this prerequisite is necessary but not sufficient. Some routines simply pass a value of T to other routines and are not true operations of T. Many routines have more than one parameter type so that it is necessary to decide which one they belong to. For all kinds of routines which convert one type into another type, this can be hard to judge. Sometimes – especially in programming languages that do not provide record types such as Fortran77 – one has to look at several base types of the underlying programming language in a parameter list to form one abstract data type. For example, one can have a stack implementation that passes two parameters, one for the stack contents realized by an array and one for the stack pointer implemented by an integer type.*

*Similarly, an abstract data object represents an abstraction of a state and the operations that manipulate the state. The state is implemented by a set of global objects. These objects are set and used by operations of the abstract data object. Most of the time, programmers do not make the effort to group the global objects of an ADO together as components of a record structure to make the connection of the object obvious. In many old programming languages they even would not have a chance to do so because user-defined data types are not supported. Even in programs written in modern programming languages, one often finds accesses to these global objects by routines that do not belong to the ADO because of efficiency considerations. All that makes it difficult to find the objects that together make up the abstract state and the routines that really represent the ADO's operations.”*

---

## ***1.1 Task Formulation***

The task of this thesis was to implement four approaches using concept analysis to identify logical components in legacy code. Four papers on the approaches had to be read and understood. The study of the original papers identified some omissions, which needed to be resolved. Then, the refined algorithms had to be implemented.

The approaches were to be evaluated using the Bauhaus benchmarks. These benchmarks allow to compare the results of the approaches with other approaches. The results of the analyses were to be evaluated and discussed.

An integration of the concept analysis approaches in the interactive tool used by Bauhaus was part of the work.

---

## ***1.2 Overview***

The thesis is structured according to the actual working process.

First, the system in which my work is embedded and tools I use are briefly described in Chapter 2.

Then, the mathematical background of concept analysis is presented in Chapter 3. The approaches and some decisions I made to fill omissions in the papers are described.

In Chapter 4 the design of the implemented units is given. The structure of the Bauhaus system is described and components for the lattice building process are presented. The interfaces between the concept analysis approaches and the lattice building are shown.

The implementation of the approaches is explained in Chapter 5. The process of analyzing a lattice is described.

Benchmarks performed with the approaches are given and discussed in Chapter 6. The results are compared with other component detection approaches.

In Chapter 7 some conclusions are made considering the benefits gained by using concept analysis for component detection.



- Bauhaus Structure
- Rigi-Editor
- concepts

---

The Bauhaus system and two tools used for this thesis – Rigi and concept – are described briefly in this chapter.

---

## **2.1 Bauhaus**

The Bauhaus project is a research cooperation of the department *Programmiersprachen und Compiler* at the *Institut für Informatik, Universität Stuttgart*, and the *Fraunhofer Einrichtung für Experimentelles Software-Engineering (FhG IESE)*, Kaiserslautern.

The Bauhaus project explores various techniques to increase the understanding of a system. The techniques are used interactively by a maintenance engineer. New techniques for modularizing legacy code semi-automatically are researched. The maintenance engineer is the center of the modularization process; he controls the process of finding components using his knowledge.

IML (InterMediate Language) is designed to represent different procedural programming languages (although only C programs are currently mapped onto IML) by offering general primitive concepts to model the semantics of all language constructs explicitly. On the other hand, by semantic annotations and specialization of modeling concepts for specific language constructs, IML allows to re-generate the original (preprocessed) source.

Using the IML-code, a *resource flow graph* (RFG) is derived. Base nodes of the graph are subprograms, user-defined types, and (global) variables, for example. The relations between the nodes are represented by edges. A reduced entity-relationship diagram for RFGs is shown in Figure 2-1. The entities are basic nodes of the RFG. The relations between the entities are realized as edges between the basic nodes. For example, the relation *subprogram p calls subprogram*

$q$  is represented by a *call\_edge* edge between the subprogram nodes *subprogram p* and *subprogram q*.

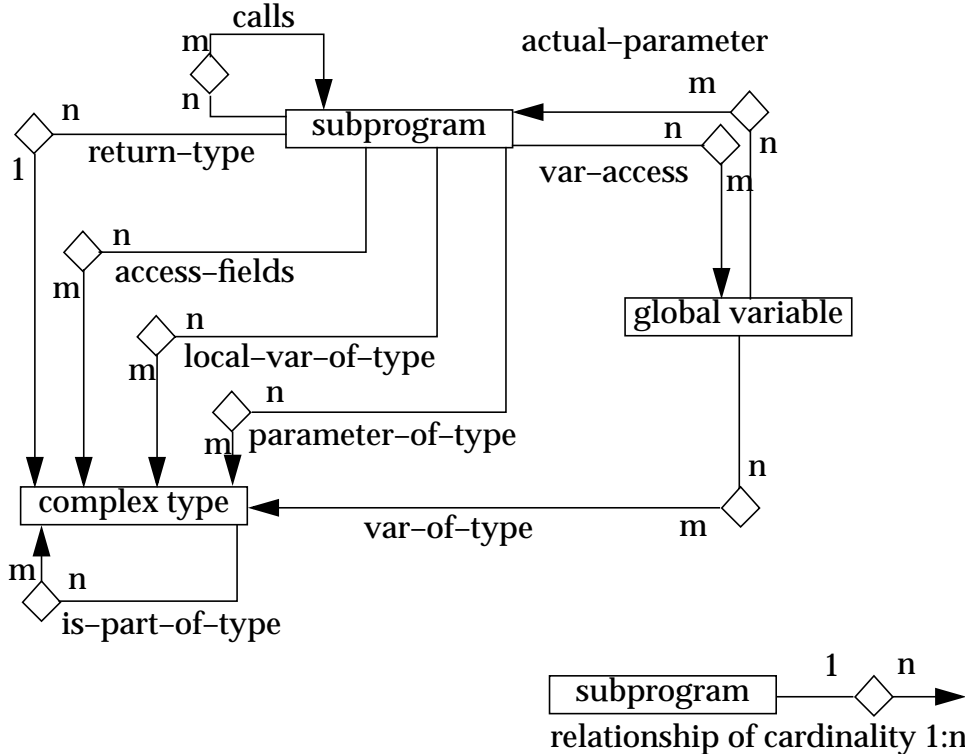


Figure 2-1. **Reduced Entity-Relationship Diagram for RFGs**

The RFG is used as a data base for the identification and visualization of architectural components. A Bauhaus analysis takes a view as input. The result of the analysis is a view containing module candidates. The result view may be used as input to the next analysis. Hence the analyses can be used incrementally. The maintenance engineer has a variety of analysis techniques to choose from. He rates the results, validates and accepts or rejects them and chooses appropriate techniques to continue the analysis. He may intersect the results of two analyses to increase the confidence in a component.

The quality of the modularization depends on the knowledge of the maintenance engineer. The Bauhaus system is a tool for faster and easier analysis with different methods.

An RFG contains some special views. All nodes that can be directly computed from the original C-code are visible in the *Base\_View*. Nodes derived from libraries are contained in the *Environment\_View* in addition to their entries in the *Base\_View*. Accepted components can be found in the *User\_View*. The parts of these components cannot be re-grouped by preceding techniques.

More information on Bauhaus can be found at [bauhaus].

RFGs are visualized by a graph editor. The editor is a modified version of Rigi.

---

## 2.2 *Rigi*

The tool for visualizing the graph information of the resource flow graph and for starting various kinds of analyses on the graph is called **Rigi**. The Bauhaus project uses a modified version of Rigi developed at the Department of Computer Science at the University of Victoria, Columbia by a research group led by Dr. Hausi Müller. The modified version of Rigi is called **rigiedit**.

Rigi is an interactive, visual tool designed to help the user to a better understanding and re-documentation of his software. The two main goals of Rigi are

1. to provide an infrastructure for research and practice in program understanding, and
2. to discover abstractions in large software systems, and pass this information on to software engineers for maintenance and reengineering purposes.

The general Rigi graph model, and the supporting graph editor, rigiedit, arose as a result of the early work of the group on algorithms for the analysis, representation, and visualization of software structure. Software structure refers to a collection of artifacts that software engineers use to form mental models when designing, documenting, or analyzing systems. These artifacts include software components – such as subsystems, procedures, variables, calls, data accesses, and interfaces – and dependencies among components – such as client-supplier, composition, and control and data-flow relations – and attributes – such as component type, interface size, and interconnection strength.

Currently, the group is focusing on visualization support to aid in the understanding and reverse engineering of legacy systems. This support is embodied in the form of a general Rigi graph model, and realized in an editor called rigiedit. Inherent in the model is the notion of nested subsystems that encapsulate detail, providing high level overviews of software systems. Recent work has generalized both the model and the tool, thus allowing the use of the graph editor in other domains, and user defined extensions to the built-in Rigi Command Library.

More information on Rigi is available at the homepage of the research group [rigi].

---

## **2.3 concept**

A useful tool to build a lattice is the tool **concept**. It was developed by Christian Lindig. I used the version `concepts-0.3d`, available at [concepts]. Because an implemented software component of mine is called `concepts`, I will use a different font to refer to the tool `concept` of Christian Lindig.

The tool `concept` reads and generates files. The input file contains the context, the output file the lattice. The syntax of the input file is simple and the format of the output file can be varied by the user. An example for an input file for `concept` is shown in Figure 2-2.

---

```
set_counter:  param_cv use_list;
init_list:    ret_list use_list;
add_item:    param_list use_list;
del_item:    param_list use_list;
num_of_items: ret_cv use_list;
```

Figure 2-2. **Small Example for a concept Input File**

---

The output format may be chosen by the option `-f`. A call of `concept` to extract the values of the nodes might be for example:

```
./concepts -olattice.txt -c -f "%i%nOBJECTS:%t%O%nATTRIBUTES:%t%A%n" lattice.con
```

The result of this program call using the example from Figure 2-2 is shown in Figure 2-3. The number of a node consists of three digits. All objects and attributes assigned to the nodes are listed.

---

```
000
OBJECTS:  num_of_items del_item add_item init_list set_counter
ATTRIBUTES: use_list
001
OBJECTS:  num_of_items
ATTRIBUTES: use_list ret_cv
002
OBJECTS:  del_item add_item
ATTRIBUTES: use_list param_list
003
OBJECTS:  init_list
ATTRIBUTES: use_list ret_list
004
OBJECTS:  set_counter
ATTRIBUTES: use_list param_cv
005
OBJECTS:
ATTRIBUTES: use_list ret_cv param_list ret_list param_cv
```

Figure 2-3. **Result of Program Call**

---

A part of the program I wrote has the function to write objects and attributes to a file in the syntax of concepts. It calls the tool `concept` twice. The first call produces a file like that shown in Figure 2-3. From this file the information about the concepts is gathered. The second file contains information regarding the relations of the concepts. This information is used to construct a lattice. Calling the tool twice is not efficient, because the main part of the analysis is spent on building the lattice. Thus, either the tool `concept` will be changed to produce both files or the second file is chosen and the information on the concepts will be computed in future.

An example for the output of the second file is shown in Figure 2-4. The example input from Figure 2-2 is used. The call of `concept` has the format:

```
./concept -olattice.gv -G lattice.con
```

The resulting file of the program call from above can be visualized by a tool called *graphplace*. This tool was used by Christian Lindig and Gregor Snelting. Because RFGs will be visualized by `rigiedit` within `Bauhaus`, the format will only be used to gather the information on the relation of the concepts.

---

```
%!PS
%% number of objects: 5
%% number of attributes: 5
%% number of concepts: 6

() () (use_list) 0 node
(num_of_items) () (ret_cv) 1 node
(del_item add_item) () (param_list) 2 node
(init_list) () (ret_list) 3 node
(set_counter) () (param_cv) 4 node
() () () 5 node

0 1 edge
0 2 edge
0 3 edge
0 4 edge
1 5 edge
2 5 edge
3 5 edge
4 5 edge
```

Figure 2-4. **Result of Program Call**

---



# *Finding Components via Concept Analysis*

- Finding ADOs and ADTs
- Graph Decomposition
- Interferences and Resolution
- Partitions
- Overlaps

---

In the following, the mathematical theory of concept analysis is introduced. Four approaches on finding logical components via concept analysis will then be presented. We will see that – although each approach uses concept analysis – they differ in considered relations and interpretation of the lattice. Two of these approaches focus on finding ADTs (the approaches of Siff/Reps and Graudejus), the other two on identifying ADOs (the approaches of Lindig/Snelting and Sahraoui et. al.). Each of them uses different kinds of attributes for the concept. Mostly they propose to try different kinds of attributes or even merge attributes.

All authors use the term “uses” or “is used by” when speaking of relations between objects and attributes. Mostly the objects are subprograms of the legacy code and the attributes “uses global variable” or “uses (complex) type”. It should be safe to assume that “uses” means at least “reads the value of” or “sets the value of”. One could also add “takes the address of” for dereferencing a variable or accessing the component of a complex type. In the introducing part of each of the different approaches, an appropriate definition of the term “uses” will be discussed.

<b>authors</b>	<b>relation(s)</b>	<b>focus</b>
Lindig/Snelting	subprogram uses global variable	ADO
Sahraoui et al.	subprogram uses (refers to) global variable	ADO
Siff/Reps	subprogram returns or has parameter of (complex) type or uses fields of (complex) type	ADT
Graudejus	v. Siff/Reps	ADT

### 3.1 Mathematical Background

The mathematical foundation of concept analysis was laid by G. Birkhoff in 1940 [Bir40]. Concept analysis provides insight into every kind of binary relations. Thus, the application of concept analysis is not bound to computer science.

One attempt to detect ADTs and ADOs can be based on inspecting the relations between subprograms and user-defined data types or global variables.

The base of concept analysis is a relation  $R$  between a set of objects  $O$  and a set of attributes  $A$ , hence  $R \subseteq O \times A$ . The terms *object* and *attribute* in the meaning of concept analysis can represent anything – not only data object and attribute in the meaning of object oriented programming. In the following, the terms *object* and *attribute* are used in the meaning of concept analysis.

**Definition:** context

The tuple  $C = (O, A, R)$  is called a **context**. The relation  $oRa, o \in O \wedge a \in A$  means that “the object  $o$  has the attribute  $a$ ”.

**Definition:** common attributes/common objects

For a set of objects  $O \subseteq O$ , the set of **common attributes**,  $\sigma$ , is defined as  $\sigma(O) = \{a \in A \mid \forall o \in O : oRa\}$ .

For a set of attributes  $A \subseteq A$ , the set of **common objects**,  $\tau$ , is defined as  $\tau(A) = \{o \in O \mid \forall a \in A : oRa\}$ .

A small example will be used to explain the theory. Let people be the objects and characteristics be the attributes of the context. The relation table in Figure 3-1 shows a possible relation between people and their characteristics. An entry X in the table means that the person of the row has the characteristic in its column.

objects/attributes	small	blond	sensitive	sporting
Samuel	-	X	-	-
Laura	X	X	-	-
Michael	X	-	X	X
Kelly	-	-	X	X
Robert	X	X	-	-

Figure 3-1. Example for a Relation Table

The following equations hold for the relation table presented in Figure 3-1  
 $\sigma(\{Michael, Kelly\}) = \{sensitive, sporting\}$  and  
 $\tau(\{small, blond\}) = \{Laura, Robert\}$ .

**Definition:** concept/extent/intent

A pair  $c = (O, A)$  is called a **concept**, iff  $A = \sigma(O) \wedge O = \tau(A)$ .

$O$  is called the **extent** of  $c$ , denoted by  $extent(c)$ .  $A$  is called the **intent** of  $c$ , denoted by  $intent(c)$ .

The table in Figure 3-2 shows all concepts of the example in Figure 3-1. They are listed starting with a **top element** that contains all objects ( $c_0$ ) and ending with a **bottom element** that contains all attributes ( $c_6$ ). Usually – but not necessarily – the top element has an empty intent and the bottom element has an empty extent.

$c_0$	$(\{Samuel, Laura, Michael, Kelly, Robert\}, \emptyset)$
$c_1$	$(\{Laura, Michael, Robert\}, \{small\})$
$c_2$	$(\{Samuel, Laura, Robert\}, \{blond\})$
$c_3$	$(\{Michael, Kelly\}, \{sensitive, sporting\})$
$c_4$	$(\{Laura, Robert\}, \{small, blond\})$
$c_5$	$(\{Michael\}, \{small, sensitive, sporting\})$
$c_6$	$(\emptyset, \{small, blond, sensitive, sporting\})$

Figure 3-2. **Concepts for Figure 3-1**

---

The set of all concepts of a given context forms a partial order via:

$$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2 \text{ or equivalently } (O_1, A_1) \leq (O_2, A_2) \Leftrightarrow A_1 \supseteq A_2.$$

**Definition:** subconcept/superconcept

If  $c_1 \leq c_2$  holds, then  $c_1$  is called a **subconcept** of  $c_2$  and  $c_2$  is called a **superconcept** of  $c_1$ . For instance,  $c_6 \leq c_5 \leq c_3 \leq c_0$  is true in Figure 3-2.

The set of all concepts of a given context,  $L$ , and the partial order  $\leq$  form a complete lattice, called **concept lattice**:

$$L(C) = \{(O, A) \in 2^O \times 2^A \mid A = \sigma(O) \wedge O = \tau(A)\}$$

**Definition:** infimum/supremum

The **infimum** of two concepts in the concept lattice is computed by intersecting their extents as follows:  $(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$ . The infimum describes a set of common attributes of two sets of objects. Similarly, the **supremum** is determined by intersecting the intents of the concepts:

$(O_1, A_1) \vee (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2)$ . The supremum describes a set of common objects of two sets of attributes.

For example: the infimum of  $c_1$  and  $c_3$  is  $c_5 = c_1 \wedge c_3$ . The supremum of  $c_4$  and  $c_5$  is  $c_1 = c_4 \vee c_5$  in Figure 3-1.

A possible representation of the concept lattice is a directed graph. Nodes in the graph represent concepts of the context. The edges denote the subconcept/superconcept relation  $<$  as shown in Figure 3-3. The relation  $<$  means:

$$c_0 < c_1 \Leftrightarrow \text{extent}(c_0) \subset \text{extent}(c_1) \text{ or equivalently } c_0 < c_1 \Leftrightarrow \text{intent}(c_0) \supset \text{intent}(c_1).$$

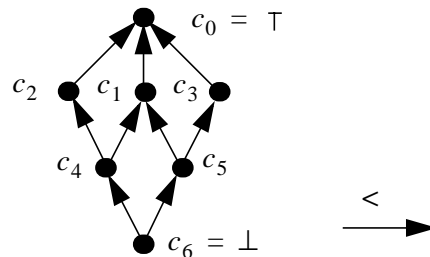


Figure 3-3. **Lattice for Figure 3-1**

---

The graph is acyclic because the relation  $<$  is chosen. The most general concept is the top element. The top element is denoted by  $\top$ . The most special concept is the bottom element, denoted by  $\perp$ . The graphic representation in Figure 3-3 and the table from Figure 3-2 together form the concept lattice.

The information, given by the concept lattice, can be represented in a reduced way. It is equivalent to the representation from above but more readable. In the new graph representation the concepts are marked with objects and attributes. The most general concept which has the attribute  $a \in A$  in its intent will be marked with it. Equivalently, the most specialized concept which has the object  $o \in O$  in its extent will be marked with it.

The unique element  $\mu$  in the concept lattice marked with  $a$  is therefore:

$$\mu(a) = \bigvee \{c \in L(C) \mid a \in \text{intent}(c)\}.$$

The unique element  $\gamma$  marked with  $o$  is:

$$\gamma(o) = \bigwedge \{c \in L(C) \mid o \in \text{extent}(c)\}.$$

A graph representing a concept lattice using this marking strategy will be called a **sparse representation**. The equivalent sparse representation for Figure 3-1 is shown in Figure 3-4.

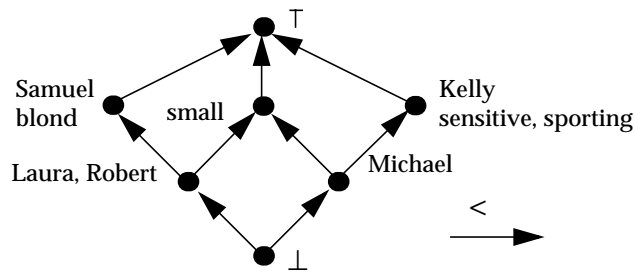


Figure 3-4. **Sparse Representation of Figure 3-1**

The content of a node  $N$  in this representation can be derived as follows:

- the objects of  $N$  are all objects at and below  $N$ ,
- the attributes of  $N$  are all attributes at and above  $N$ .

For instance, the node in Figure 3-4 marked with *Michael* is the concept  $c_5 = (\{Michael\}, \{small, sensitive, sporting\})$ .

The sparse representation is generated by the tool *concept* of Christian Lindig in the standard output format.

## 3.2 Lindig and Snelting

Christian Lindig and Gregor Snelting investigate the relations between subprograms and global variables [LinSnel97]. They are focusing on detecting ADOs. In this section, their work on horizontal decomposition by resolving interferences in the context lattice will be presented. In this approach, objects are subprograms and an attribute is “uses global variable  $v$ ”. It seems appropriate to assume that “uses” means “reads” or “sets”. Maybe “takes the address of” is accepted additionally. Two possibilities for modularizing a lattice are pointed out: modularization by horizontal decomposition or via block relations. Block relations are only described briefly here. This approach is not implemented. The approach via horizontal decomposition is presented in detail. For this approach

an algorithm is proposed. The idea of this algorithm is implemented by identifying horizontal summands and resolving the interferences between these horizontal summands.

Let  $C = (O, A, R)$  be a context in the meaning of concept analysis.  $O$  is the set of objects,  $A$  is the set of attributes, and  $R$  is the relation between objects and attributes. In this approach the set of subprograms  $P$  is equal to the objects  $O$  and the set of variables  $V$  is equal to the attributes  $A$ .

Lindig and Snelting present their own definition of an abstract data object (ADO). This definition is only valid for this section (3.2).

**Definition:** abstract data object (ADO)

An ADO consists of a set  $P \subseteq P$  and a set  $V \subseteq V$ , such that

$$(\forall p \in P, v \in V : (p, v) \in R \rightarrow p \in P) \wedge (\forall p \in P, v \in V : (p, v) \in R \rightarrow v \in V).$$

In words: all variables in  $V$  are only used by subprograms in  $P$  and all subprograms in  $P$  use only variables in  $V$ .

**Definition:** common/used variables/subprograms

Let  $P \subseteq P$  be a set of subprograms. Then the common variables of  $P$  are

$$cv(P) = \{v \in V \mid \forall p \in P : (p, v) \in R\}.$$

Let  $V \subseteq V$  be a set of global variables. Then the common subprograms of  $V$  are  $cp(V) = \{p \in P \mid \forall v \in V : (p, v) \in R\}$ .

Let  $P \subseteq P$  be a set of subprograms. Then the used variables of  $P$  are

$$uv(P) = \bigcup_{p \in P} cv(\{p\}).$$

Let  $V \subseteq V$  be a set of global variables. Then the used subprograms of  $V$  are

$$up(V) = \bigcup_{v \in V} cp(\{v\}).$$

The common and used variables and subprograms will be used to define submodules and interference of concepts. The common subprograms of  $V$  are the subprograms that each use all variables of  $V$ . The used subprograms of  $V$  are all subprograms that use variables of  $V$ . The common variables are a subset of the used variables. For example, the common variables of  $P = \{subp2, subp5\}$  are  $cv(P) = \{var2, var4\}$  in the variable usage table of Figure 3-5. The used variables of  $P = \{subp2, subp5\}$  are  $uv(P) = \{var2, var3, var4\}$ .

**Definition:** variable usage table

Let  $P$  be a set of subprograms. Let  $V$  be a set of variables. A variable usage table is a relation  $R \subseteq P \times V$ . If and only if the subprogram  $p \in P$  uses global variable  $v \in V$  then  $(p, v) \in R$ .

subprograms/variables	var1	var2	var3	var4
subp1	-	-	X	-
subp2	-	X	-	X
subp3	X	-	X	X
subp4	X	-	-	X
subp5	-	X	X	X

Figure 3-5. Example for a Variable Usage Table

### 3.2.1 Block Relations

Block relations are based on maximal rectangles in the variable usage table. The variable usage table displays the relation between objects and attributes. An example for such a table can be found above in Figure 3-5. A maximal rectangle in the variable usage table can be constructed by replacing empty entries with a  $o$ . If, for example, in the table in Figure 3-5 a maximal rectangle is created, one could replace the entries  $(subp3, var2)$  and  $(subp2, var3)$  such that a maximal rectangle would be  $(\{subp2, subp3, subp5\}, \{var2, var3, var4\})$  as can be seen in Figure 3-6. The process of creating such maximal rectangles is not described in detail in the article of Christian Lindig and Gregor Snelting.

subprograms/variables	var1	var2	var3	var4
subp1	-	-	X	-
subp2	-	X	o	X
subp3	X	o	X	X
subp4	X	-	-	X
subp5	-	X	X	X

Figure 3-6. Example for a Maximal Rectangle

Each maximal rectangle in the variable usage table represents a module candidate. These rectangles are called block relations.

**Definition:** block relation (rectangle shape)[LinSne97, p 355]

Let a formal context  $C = (P, V, R)$  be given. A block relation is a formal context  $C' = (P, V, R')$  where  $R \subseteq R'$ , and for  $p \in P$ ,  $cv_{R'}(p)$  is an extent in  $L(C)$ , and for  $v \in V$ ,  $cp_{R'}(v)$  is an intent in  $L(C)$ .

**Definition:** module

Let  $P \subseteq P$  and  $V \subseteq V$ . Then  $(P, V)$  is called a module iff  $uv(P) \subseteq V \wedge up(V) \subseteq P$ .

This means:  $V$  contains at least all variables used by subprograms in  $P$  and  $P$  contains at least all subprograms using variables of  $V$ .

**Definition:** submodule

Let  $(P, V)$  be a module and  $S \subseteq P$ . A module  $(S, V)$  is called a submodule of  $(P, V)$  iff  $uv(P \setminus S) \neq V$ .

Hence a submodule consists of a subset of the modules subprograms and may have additional variables. These additional variables are global variables but are only accessed by the submodules subprograms. Thus these variables could be called “local” variables of the submodule.

**Definition:** local subprograms

Let  $q \in P$  and  $p \in P$ . Then  $q$  is called local to  $p$  iff  $cv(p) \subseteq cv(q)$ .

This definition is based on the assumption, that a local subprogram introduces its own set of state variables, which cannot be used by the subprograms on higher levels.

The method will only work if the module candidates found in the variable usage table represent regular cohesive modules.

**Definition:** maximal/regular cohesion

An ADO  $(P, V)$  has maximal cohesion iff  $\forall p \in P, v \in V: (p, v) \in R$ .

An ADO  $(P, V)$  has regular cohesion iff

$(\exists \bar{p} \in P: \forall v \in V: (\bar{p}, v) \in R) \wedge (\exists \bar{v} \in V: \forall p \in P: (p, \bar{v}) \in R)$ .

Most of the modularizations which may be computed from legacy code do not have maximal cohesion. Thus regular cohesion often has to suffice for component finding. As an example for  $\bar{p}$  one could examine an initializing subprogram. An initializing subprogram typically accesses all variables of an ADO. A reading subprogram or a writing subprogram are other possible examples. Finding an example for  $\bar{v}$  is harder. It is not typical for an ADO to contain a variable that is accessed by all subprograms of the ADO. As the example in Figure 3-7 shows, not all module candidates will be accepted by applying regular cohesion. There is no  $\bar{p}$  or  $\bar{v}$  which fulfills the regular cohesion, but the subprograms and their referenced variables might be candidates for a logical module. One could define edge cohesion as another kind of cohesion, which would accept the example in Figure 3-7.

**Definition:** edge cohesion

An ADO  $(P, V)$  has edge cohesion

iff  $(\forall p_1, p_2 \in P \exists v \in V: p_1 R v \wedge p_2 R v) \vee (\forall v_1, v_2 \in V \exists p \in P: p R v_1 \wedge p R v_2)$ .

This means that either each pair of subprograms shares a variable or each pair of variables shares a subprogram (“subprogram uses variable” – relation). In the example shown in Figure 3-7 the subprograms  $P_1, P_2$  share the variable  $V_2$ .

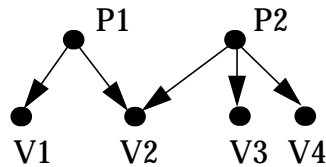


Figure 3-7. **Small Example for Cohesion**

The method using block relations was proposed by Lindig and Snelting because of the overlapping sublattices resulting from nested subprograms. Subprograms usually do not use all variables of an ADO. Thus horizontal decomposition of the lattice will often not be possible.

### 3.2.2 Horizontal Decomposition

A possible horizontal decomposition of a lattice indicates an easy modularization of the legacy code. Interferences inhibit a horizontal decomposition. A horizontal decomposition is the inverse to a horizontal sum. The horizontal sum of summand lattices  $L_1, L_2, \dots, L_n$  is

$$\sum_{i=1}^n L_i = \{T, \perp\} \cup \bigcup_{i=1}^n L_i \setminus \{T_i, \perp_i\}.$$

That is, the local top and bottom elements are removed from each  $L_i$ , and new global top and bottom elements are added. Conversely, a lattice  $L$  is horizontally decomposable, if it is a horizontal sum. The module corresponding to a horizontal summand  $L_i$  is  $(P, V) = (ext(\{T_i\}, int\{\perp_i\}))$ . A horizontal decomposition identifies ADOs.

A so-called **interference** inhibits a horizontal decomposition. Thus it has to be resolved. An interference occurs if a concept uses variables (in the meaning of the relation of the context) of different module candidates (would-be horizontal summands). The interference is the infimum of these module candidates. An example for an interference can be found in Figure 3-8.

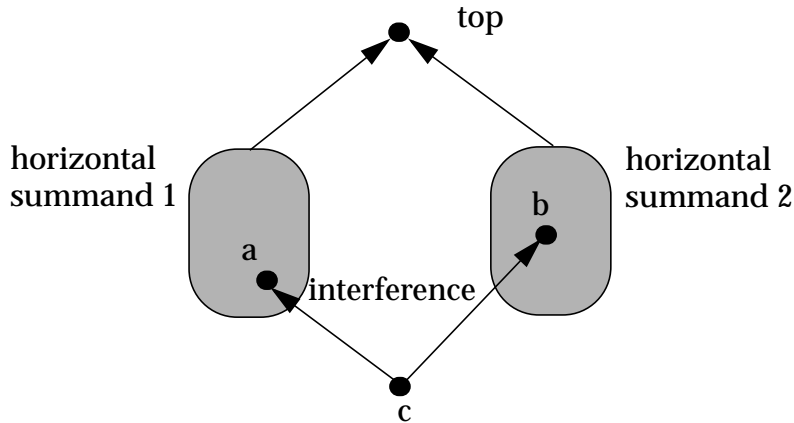


Figure 3-8. **A Simple Interference**

**Definition:** interference

$V_1, V_2 \in V$  interfere via  $p \in P$  iff  $p \in up(V_1) \cap up(V_2)$ .

An interference is resolved by choosing one of the superconcepts and deleting the edge between sub- and superconcept. The edge is eliminated by logically inserting the colliding uses (attributes) as parameters of the subprogram (the object). This only deletes the entry from the variable usage table and thereby the edge from the lattice. Let  $c$  be the subconcept and  $a, b$  be the superconcepts of  $c$ ;  $a$  and  $b$  interfere. If  $|int(c) \setminus int(a)| > |int(c) \setminus int(b)|$ , then the edge from  $c$  to  $a$  will be deleted from the lattice because  $c$  inherits more variables from  $b$  than from  $a$ . If  $|int(c) \setminus int(a)| < |int(c) \setminus int(b)|$ , then the edge from  $c$  to  $b$  will be removed. In the article is no hint what to do if  $|int(c) \setminus int(a)| = |int(c) \setminus int(b)|$ , so I will choose the superconcept to be unchanged by means of the following rules in their listed order. A concept will not be changed if

- i) it has the higher cardinality of  $ext(c)$
- ii) it has the lower cardinality of  $int(c)$
- iii) it is chosen randomly.

The interference analysis algorithm proposed by Lindig and Snelting is supposed to decompose the lattice in such a way, that the connectivity will be minimal. Thus the cohesion will be maximal and the coupling minimal [Sne95, p 21].

**Definition:** connectivity  $k$

The connectivity  $k$  of a lattice counts the edges between sublattices.

---

### interference analysis algorithm

1. Try to find a horizontal decomposition of the lattice. For this purpose remove the top and bottom element and try to find connected components. If successful, there are no interferences on the top level (connectivity  $k = 0$ ). The algorithm is then applied recursively on the sublattices each containing the removed top and bottom element as new top and bottom.
2. If the decomposition fails, there may be interferences. Detect interferences of connectivity  $k = 1$  by removing the top and bottom element and computing the biconnected components of the remaining graph. A bridge between two biconnected components which leads to an  $\wedge$ -reducible concept (of the form  $c = a \wedge b$ ) points to an interference. Highlight the node  $c$ .
3. Because there is often more than one interference, compute the  $k$ -connected components of the graph, where  $k$  is minimal. A method to determine  $k$ -connected sublattices is to consider all sets of  $k$   $\wedge$ -reducible concept nodes and test whether their removal will break the graph into unconnected subgraphs.

Algorithm 3-1. **Algorithm for Minimal Interference Resolution** [Sne95, p 21]

---

The term  $\wedge$ -reducible points to the possibility of deducing  $c$  of  $a$  and  $b$  by building the infimum  $a \wedge b$ . Thus  $c$  is a subconcept of  $a$  and of  $b$ .  $k$   $\wedge$ -reducible points to a concept with  $k$  superconcepts from which  $c$  can be deduced by applying  $a_1 \wedge a_2 \wedge \dots \wedge a_k$ .

---

### 3.3 *Sahraoui et al.*

Houari A. Sahraoui, Walcéllo Melo, Hakim Lounis and François Dumont work on transforming legacy procedural code into object-oriented code [Sah97] and concentrate on detecting ADOs as a first step. Their approach uses concept analysis to find candidate objects and identify the objects using the candidate objects. Afterwards they group the subprograms as methods of the objects. The term object is used by Sahraoui et. al. in the meaning of object-oriented programming. So the candidate objects are not candidate objects in the meaning of concept analysis. Thus I will call the object-oriented term DO (for data object).

In contrast to usual practise, a variable is an object and a subprogram using a variable is an attribute. This originates in the object-oriented approach. The global variables will become the DOs and the attributes “is used by subprogram”

will lead to association of subprograms to methods of DOs. The approach discriminates different modes of “using” [Sah97, p 213]:

- modification or write mode (m) when the subprogram modifies the value of the variable,
- access or read mode (a) when it uses its value to compute something else, and
- predicate mode (p) when the variable is used to control the execution of the subprogram (in a predicate).

In the approach of Sahraoui et. al., a reference graph for the usage information is generated. Afterwards the relations are extracted and a concept lattice is built. The relation *refers-to*( $f, v, t, m$ ) has the values  $f$  for a subprogram,  $v$  for the global variable,  $t$  for the type of the variable and  $m$  for the usage mode. The type information would not be needed in the relation, because it is constant for a variable. The type information can be accessed via the variable.

An algorithm identifies DO candidates in the lattice. The algorithm runs bottom-up in the lattice because of the hypothesis that DO candidates consist of a group of variables which are simultaneously accessed by a large number of subprograms. Remember, that Sahraoui et. al. assign subprograms as attributes and global variables as objects. Therefore concepts with many subprograms can be found near the bottom of the lattice. These candidate objects are used to build a new lattice. A second algorithm tries to find DOs in this candidate lattice. Afterwards the methods are identified by assigning subprograms to the DOs found in the candidate lattice.

### 3.3.1 DO Identification

Let  $O$  be the set of global variables,  $CO$  the set of candidate DOs and  $NS$  the set of not-yet-selected objects. Let  $WL$  be a working list of concepts to be examined for candidate DOs and  $L(C)$  the set of concepts of a formal context  $C$ . Algorithm 3-2 is used to identify DOs. The subprogram *next-candidate* is the tricky part of this algorithm. There are several criteria to ascertain the next candidate DO. If a concept  $c$ , which would be the next one, contains only one variable of a basic type, it is ignored. If a concept  $c$  has the predicate  $extent(c) \cap NS = \emptyset$  it will be ignored, too. The concept with the most subprograms is chosen. If there is a tie, the concept is chosen which has less variables. If there is still a tie, the concept is chosen which has the higher cardinality of the set  $extent(c) \cap NS$ ; this means which contains more variables still to be selected.

Sahraoui et. al. seem to assume that there are no DOs containing only one variable of a basic type such as a counter or a clock. If the legacy code contains such

**candidate object identification** $NS = O$  $CO = \emptyset$  $WL = L(C)$ **while**  $NS \neq \emptyset$  **do** $c = next-candidate$  $NS = NS - extent(c)$  $CO = CO \cup intent(c)$  $WL = WL - c$ **endwhile**Algorithm 3-2. **Algorithm for the Identification of Candidate DOs** [Sah97, p 214]

a DO, it will be ignored while searching for candidate DOs. Complex DOs contain more than one variable and thus will be detected.

For the object identification, let  $O'$  be the set of candidate DOs and  $A'$  the set of global variables. The relation  $R'$  is interpreted such that  $\forall g \in O' \forall v \in A', gR'v$  means  $v \in extent(g)$ . The second lattice is constructed, using the new context  $C' = (O', A', R')$ . I found a way to inhibit the second lattice building process. In Section 5.3.2 a method is described using a copy of the lattice containing only the DO candidates. Beginning at the top concept, the concepts are inspected. If a concept is chosen as DO candidate, it is copied to the new lattice. It may happen, that a concept is not chosen, but subconcepts of it are. The DO candidates in the new lattice then are in need of successors. Thus, the successors of their unchosen subconcept are taken as new successors.

Afterwards DO candidates are merged if their variable sets overlap in two variables or more. This number seems to be arbitrarily fixed in their prototype. The condition of merging will be altered in the implemented version of this approach. I will merge two concepts if they overlap in 50% or more of their variables. This approach is arbitrary too, but I hope that it will lead to better results. The merging of concepts will be a bottom-up approach, leading to the DOs in the topmost level of the second lattice. These concepts will then be chosen and presented as DOs. They only lack their methods, so this will be the next step: identifying the appropriate methods for the DOs.

**3.3.2 Method Identification**

Sahraoui et. al identify methods by inspecting the subprograms of the legacy code and applying three rules [Sah97, p 215f]. Two defined operations are

needed to describe those rules in few words. For these definitions let  $Obj$  be the set of identified DOs,  $Subp$  the set of subprograms in the legacy code, and  $Var$  the set of global variables.

**Definition:**  $ref(p)$

$\forall s \in Subp, ref(s) = \{o \in Obj | \exists v \in Var \wedge v \in extent(o) \wedge vRs\}$ , where  $R$  is the relation “is used by”.

**Definition:**  $modif(p)$

$\forall s \in Subp, modif(s) = \{o \in Obj | \exists v \in Var \wedge v \in extent(o) \wedge vMs\}$ , where  $M$  is the relation “is modified by” The relation  $M$  is derived from the relation  $R$  with the condition that the mode of usage is  $m$ .

**Rule 1:** For a subprogram  $s$ , if cardinality of  $ref(s) = 1$ , then  $s$  becomes a method of the unique DO in  $ref(s)$ .

**Rule 2:** For a subprogram  $s$ , if cardinality of  $ref(s) > 1$  and cardinality of  $modif(s) = 1$ , then  $s$  becomes a method of the unique DO in  $modif(s)$ .

By this rule, a subprogram is attached to the DO it modifies. For example, if a global variable  $v$  is copied to another global variable  $w$  by a subprogram  $s$ , the subprogram  $s$  will become a method of the DO containing the global variable  $w$ .

**Rule 3:** For a subprogram  $s$ , if cardinality  $ref(s) > 1$  and cardinality of  $modif(s) > 1$ , then  $s$  must be sliced when possible to create a method for each DO in  $modif(s)$ .

In this case there seems to be a method, as for example a global initialization, which could be sliced in local initializations for each DO. If it is this easy, there might be the solution to just present a local method for each DO involved.

---

### **3.4 Siff and Reps**

Michael Siff and Thomas Reps are focusing on finding ADTs [SifRep99]. The goal of their work is to identify possible modularizations by computing partitions of the objects of the lattice. The partitions are presented to the maintenance engineer to choose the best one for modularization.

Siff and Reps are using subprograms as objects. For the attributes, different alternatives are proposed [SifRep99, p 760]. Even merging of the attributes is moti-

vated. For the tested examples, Siff and Reps admittedly used only three kinds of attributes: a subprogram “returns a value of type  $t$ ”, “has a parameter of type  $t$ ” or “uses fields of the complex type  $t$ ”. The term “uses fields of” will be interpreted as “reads the value of a field of”, “sets the value of a field of” or “takes the address of a field of”. All of those attributes are used in the same type usage table. The **type usage table** is the analogon of the variable usage table for types. In the implementation of this approach, the attribute “uses fields of” is unaccounted for. Only the attributes related to the signature of the subprogram are used.

At first the concept lattice is built. The lattice has often to be enhanced to construct a so called **well-formed lattice** with the algorithm described below.

**Definition:** well-formed context

A context is *well-formed* iff  $\forall x, y \in O, \sigma(\{x\}) \subseteq \sigma(\{y\}) \rightarrow \sigma(\{x\}) = \sigma(\{y\})$ .

In words: in the type usage table of a well-formed context there exists no row that is a real subset of another row. Rows are either different or equal.

**Definition:** well-formed lattice

A lattice is *well-formed* iff it represents a well-formed context.

The well-formed lattice guarantees **atomic concepts**. The algorithm uses these concepts as the initial worklist for the partitions. A partition of the concepts that consists only of atomic concepts is called an **atomic partition**. Hence a well-formed lattice can be partitioned by an atomic partition.

**Definition:** atomic concept

An atomic concept  $c_a$  is a concept built by using  $c_a = (\sigma(\{a\}), \tau(\sigma(\{a\})))$  where  $a \in A$ .

In the concept lattice, the atomic concepts are the concepts on the lowest level. They are created by applying the rule from above. This rule guarantees their position directly above the bottom of the lattice.

**Definition:** concept partition

Let  $C = (O, A, R)$  be a context.  $P = \{(X_0, Y_0), \dots, (X_n, Y_n)\}$  is a concept partition of the context if  $(\bigcup X_i = O) \wedge (X_i \cap X_j = \emptyset, i \neq j)$  holds. The trivial partition is  $\{(O, \emptyset)\}$ .

A **concept partition** partitions the objects of the context. In the approach of Siff and Reps, objects are subprograms. Thus partitions of the subprograms are created. The partition of the subprograms might be a problem. Generally, not all subprograms are associated to an ADT. Subprograms that do not belong to an ADT ought to be ignored by a concept partition. Identifying these subprograms

beforehand and group them by other means before applying the partitioning algorithm could enhance the process.

The partitions – computed by the algorithm – are presented to the maintenance engineer for validation. To guarantee an atomic partition, the relation table sometimes has to be enhanced with **complement attributes**.

**Definition:** complement attribute

Given a context  $C = (O, A, R)$ , a complement of an attribute  $a \in A$  is an attribute  $\bar{a}$  such that  $\tau(\{\bar{a}\}) = \{x \in O \mid (x, a) \notin R\}$ .

For the implementation, a heuristic is needed. For an automatic computation of the modularization candidates, a heuristic chooses the partition as result, which seems to contain the best candidates for modularization. Siff and Reps propose to enhance the algorithm by interacting with the user of the tool. The algorithm to construct a well-formed context and the one to compute the partitions follow.

---

construct a **well-formed lattice**

$A' \leftarrow A$

$R' \leftarrow R$

**while**  $(O, A', R')$  is not well formed **do**

**let**  $x, y \in O$  be such that  $\sigma(\{x\}) \subset \sigma(\{y\})$

**let**  $a \in A'$  be such that  $a \notin \sigma(\{x\}), a \in \sigma(\{y\})$

$A' \leftarrow A' \cup \{\bar{a}\}$ , where  $\bar{a}$  is a new attribute

$R' \leftarrow R' \cup \{(x, \bar{a}) \mid (x, a) \notin R'\}$

**endwhile**

Algorithm 3-3. **Algorithm for Making a Lattice Well-Formed** [SifRep99, p 759]

---

Where  $C = (O, A, R)$  is the original context. Look for offending pairs and add a complement attribute  $\bar{a}$  to the variable usage table, where  $a$  is part of  $\sigma(y)$  but not part of  $\sigma(x)$ .

**Definition:** covers/covs/subs

A concept  $d$  covers a concept  $c$  if  $c < d$  and there is no concept  $e$  such that  $c < e < d$ . If  $d$  covers  $c$ , we say that “ $c$  is covered by  $d$ ”.

$covs(c) = \{d \in L(C) \mid c < d \wedge \neg(\exists e \in L(C) : c < e < d)\}$

and  $subs(d) = \{c \in L(C) \mid c < d\}$ .

---

find the **partitions** of a well-formed concept lattice

```

A ← covs( $\perp$ )
P' ← {A}
W ← {A}
while W ≠  $\emptyset$  do
  remove some p from W
  foreach c ∈ p do
    foreach c' ∈ covs(c) do
      p' ← p - subs(c')
      if ( $\forall c'' \in p' : \text{extent}(c'') \cap \text{extent}(c') = \emptyset$ ) then
        p'' ← p' ∪ {c'}
        if p' ∉ P then
          P ← P ∪ {p''}
          W ← W ∪ {p''}
        endif
      endif
    endfor
  endfor
endwhile

```

Algorithm 3-4. **Algorithm for Computing all Partitions of a Context**  
[SifRep99, p 759]

---

In the approach of Siff and Reps, so-called complement attributes are created and inserted in the type usage table to guarantee an atomic partition. The number of concepts is increased by this method.

Because of the complement attributes, there may be concepts containing only complement attributes. The semantic of such a concept would be that the subprograms in the extent of the concept do not use the types in the intent of the concept. This concept is of no use in detecting ADTs. Partitions calculated by the algorithm above may contain many of these concepts. Such partitions should not be presented to the user of the system.

### 3.5 Graudejus

Holger Graudejus describes his approach on detecting ADTs as inspired by the approach of Michael Siff and Thomas Reps. He uses subprograms as objects and “has argument of type x”, “has return-value of type x” and “uses fields of a variable of type x” as attributes [Gra98]. The term “uses fields of” is still not defined, so the interpretation applied to “uses” in Siff and Reps – “reads fields of”, “sets fields of” and “dereferences fields of” – is chosen.

After building the concept lattice, ADT candidates are searched. Graudejus assumes that they are located at the top of the lattice because there the concepts with most of the subprograms and least of the attributes in the whole lattice can be found. The property of an ADT is that it consist of subprograms grouped around few data types. Thus the assumption to find good candidates in the top-most level of the lattice seems cogent.

The process of constructing the lattice is the same as the one used by Siff and Reps. In the approach of Graudejus, overlapping concepts are searched for in the lattice. An **overlap** of two concepts occurs if they share subprograms. An overlap of two concepts results in a shared subconcepts of the overlapping concepts. The extent of the subconcept contains the overlapping subprograms. This overlap is resolved by an overlap competition. The overlapping subprograms are deleted from the losing concept of this competition and the originating concept is inserted in the lattice after the loser of the competition is deleted from the lattice. The algorithm focuses on overlaps of the ceiling concepts of the lattice.

**Definition:** ceiling concept [Gra98, p 41]

Let  $c = (X_0, Y_0)$ ,  $c' = (X_1, Y_1)$  be concepts, let  $t$  be the top concept.

We call a concept  $c$  **ceiling concept** iff for all concepts  $c' \neq t$  the following holds:  $X_1 \subseteq X_0$ .

This means the ceiling concepts are the concepts located directly below the top-concept  $t$ . The condition  $X_1 \subseteq X_0$  allows only one ceiling concept per lattice. A lattice containing more than one concept on the level below the top concept does not contain ceiling concepts, applying the definition from above. Hence, the definition made by Graudejus seems to be wrong. Here is the definition how it might have been intended:

**Definition:** *ceiling concept*, as probably intended

Let  $c = (O, A)$  be a concept; let  $t$  be the top concept,  $L(C)$  the set of concepts of a context  $C$ . We call a concept  $c$  **ceiling concept** iff  $\neg \exists c' \in L(C) \setminus \{c, t\} : c \leq c'$ .

The proposed **overlap competition** of Graudejus is as follows [Gra98, p 44]: a concept loses

- i) if it contains a type that is a part type of its competitors type, and the same is not true for its competitor
- ii) else if none of the overlapping subprograms accesses internal fields of one of its types, and the same is not true for its competitor
- iii) else if none of the overlapping subprograms has one of its types as a return type, and the same is not true for its competitor.

This overlap competition already contains a combination with other techniques. The *Part Type* [LiuWil90] and the *Internal Access* [Yeh95] heuristic are applied.

The algorithm to resolve the overlaps starting with the ceiling-concepts top-down follows.

---

**ceiling-focused concept analysis**

set up the lattice

**while** two ceiling-concepts overlap **do**

    perform the overlap competition

    remove the overlapping subprograms from the loser and insert

        this concept into the concept lattice

    delete the loser from the concept lattice

**endwhile**

present the ceiling-concepts

Algorithm 3-5. **Algorithm for Resolving Overlaps** [Gra98, p 41]

---

---

### ***3.6 Combination with other Techniques***

An optional part of the thesis was to combine different techniques with the concept analysis approaches. Originally this was planned as an extra part of the thesis. During the implementation phase, I had to fill some omissions to implement the algorithms. If there was no proposal how to decide between equal candidates, I chose to use the *Internal Access* heuristic.

**Internal Access** means, that a subprogram accesses the internal structure of a variable or a type. The violation of the information hiding principle is a hint that subprograms and their accessed entities are related. A more detailed description of the heuristic can be found in [Yeh95]. A description of *Part Type* can be found in [LiuWil90].

The approach of Graudejus uses the heuristics *Part Type* and *Internal Access*. I used the *Internal Access* heuristic for the approach of Sahraoui. In my implemen-

tation of the approach of Siff and Reps, *Internal Access* is used to quantify the quality of a partition.

- Ada95
- Hierarchical Packages rfgs and acd
- Algorithms

---

In this section, the design of the algorithms and components the algorithms are based on will be presented. There are two levels of the design: the first level consists of the components to build a lattice, the second level consists of the algorithms based on the first-level components.

---

## **4.1 Overview**

The Bauhaus system is implemented in Ada95. A reference on Ada95 can be found at [ada]. Useful books on Ada95 are “The Language Reference Manual” [LRM94] and “programming in Ada95” [Bar98], for example. Ada95 allows hierarchies of library units. The components I implemented are added to the subsystem “atomic component detection” and to the RFG implementation. New packages and changes of existing ones were necessary. All of the components will be integrated in the Bauhaus system. The structure of the system dictates the design of my components. Thus, the structure of Bauhaus will be explained briefly in the next section (Section 4.2).

The lattice building process will be described in Section 4.3. The structure and design of the components that were implemented follow in Section 4.4. The components for the lattice building will each be described in detail in an own subsection. Afterwards the design of the algorithms and the way they depend on the components used for the lattice building is described briefly in Section 4.5. In Section 4.6 an incremental approach is described.

---

## **4.2 Bauhaus structure**

The Bauhaus system contains some special directories for different kinds of files. The packages of the system are grouped in two hierarchies (atomic component detection and resource flow graph rfgs) and some other packages.

Naming rules for packages exist. All packages considering RFGs are located in the directory `bauhaus/Ada-Code/RFG/src`. The packages for atomic component detection are located in `bauhaus/Ada-Code/Atomic-component-detection/src`. Some useful code intended to be reused, like lists or stacks, is located in the directory `bauhaus/Ada-Code/Reuse/src`. A reduced hierarchy of the rfg-package is shown in Figure 4-1.

---

```
rfgs
  accessors
  edges
    composites
    concepts
  io
    components
    plain
    text
  iteration
  manipulations
  nodes
    concepts
  predicates
    edges
    nodes
  traversals
  views
```

Figure 4-1. **Reduced Hierarchy of the Package rfgs**

---

The package `accessors` contains subprograms to access data of edges and nodes, for example, a function to get the successors of a node considering a specific kind of edge. With the subprograms in `iteration`, one has the possibility to iterate over all edges or all nodes of an RFG (or a `View_Set`). A subprogram – given as generic parameter – will be executed for each of the iterated entities. `Manipulations`

is needed for inserting edges or nodes in or deleting them from Views or View\_Sets. Many subprograms are generic to allow reusability.

To implement the concept analysis algorithms, some additional nodes and edges are needed. The specifications of those nodes and edges were added to bauhaus/Ada-Code/RFG/src as child units Rfgs.Nodes.Concepts (new nodes) and Rfgs.Edges.Concepts (new edges).

---

### ***4.3 The Lattice Building Process***

One part of my work was to provide a function that generates a lattice for a given context. The process of building a lattice to a given context – using a simple algorithm – is described in [SifRep99, p 751f].

The result of the lattice building process is a view in the RFG containing the concepts and their relations to each other. The process of building the lattice as a view in a given RFG is shown in Figure 4-2.

First, the function is called by the editor for RFGs – rigiedit. This call provides the Source\_View and other information needed for building the lattice. The component **build\_lattice** filters the data from the Source\_View and creates the result view in the RFG. The result view is filled with the concepts and their relations by the component **concepts**. The call of the tool concept and the handling of the input-file and the output-files of concept is encapsulated in the component **generic\_concept\_analysis**.

---

### ***4.4 Design of the Concept Analysis Components***

The lattice building process contains a part that calls the tool concept [concepts], implemented by Christian Lindig. This component – **generic\_concept\_analysis** – also deals with the files that are needed or created by concept. To have the possibility to build a concept lattice not considering the types of objects or attributes, the component maps objects and attributes to numbers. Thus it may be used by other applications as well. For the reusability it has to be generic. This component is completely independent from the RFG implementation.

The component that builds the lattice as a view of the RFG – **concepts** – uses an instance of the generic component. It offers the routines to insert concept nodes and subconcept edges used by the generic component. In the package specification the type of objects and attributes of the relation is defined.

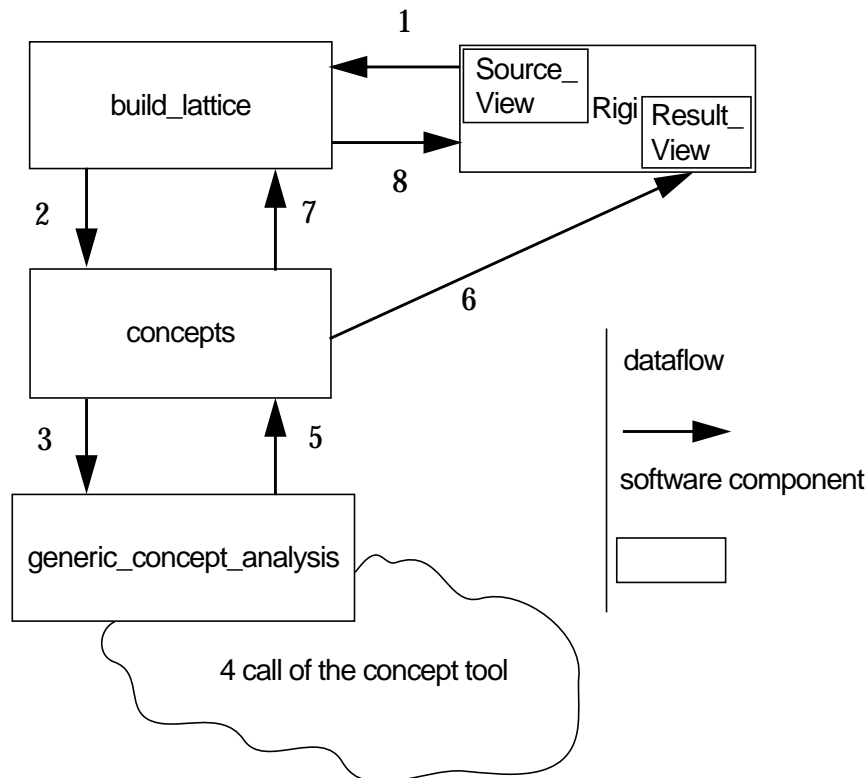


Figure 4-2. **Building a Concept Lattice**

On top of this component is the component **build\_lattice**. It filters the objects and attributes (nodes and edges) to guarantee that atomic components already accepted by the user will not be changed. These accepted components are visible in the `User_View`. The objects and attributes in the `Environment_View` are ignored. Objects and attributes for the concept analysis process are gathered from the RFG, filtered and processed by **concepts**. The result of the Perform function in **build\_lattice** is the view that contains the lattice resulting from the view given as parameter `Base_View`.

#### 4.4.1 generic\_concept\_analysis

The component that calls concept – the binary of Lindig’s tool – and handles the files is the instance of a generic package. The type of the objects and the attributes and some subprograms to deal with attributes and objects have to be given as generic parameters when an instance of **generic\_concept\_analysis** is created.

The objects and attributes are then mapped to numbers internally. Thus, whatever the objects and attributes are, the process of writing the data of the relation table to a file and reading the output of concept will always be the same. The

mapping to numbers assumes that objects and attributes are unique. Thus the mapping of objects and attributes to numbers is bijective.

The component contains a relation table that represents the relation between objects and attributes. This table has to be filled by the client unit. Afterwards the function `Build_Lattice` can be called. This method writes the contents of the relation table to a file (`lattice.con`). The tool `concept` is called once to produce a file containing the concept nodes (`lattice.txt`) and once to produce a file containing the subconcept relation between the concept nodes (`lattice.gv`). The resulting files are read and the concept nodes and subconcept edges are added by calling the routines `make_concept`, `insert_concept` and `insert_subconcept_edge` of the using module, which have to be given as parameters for the instantiation of the generic concept analysis. Thus, the lattice data need not to be duplicated. The data concerning the lattice will only be present in the client module.

The process of calling the tool `concept` and building the lattice from the context given as a relation table is shown in Figure 4-3.

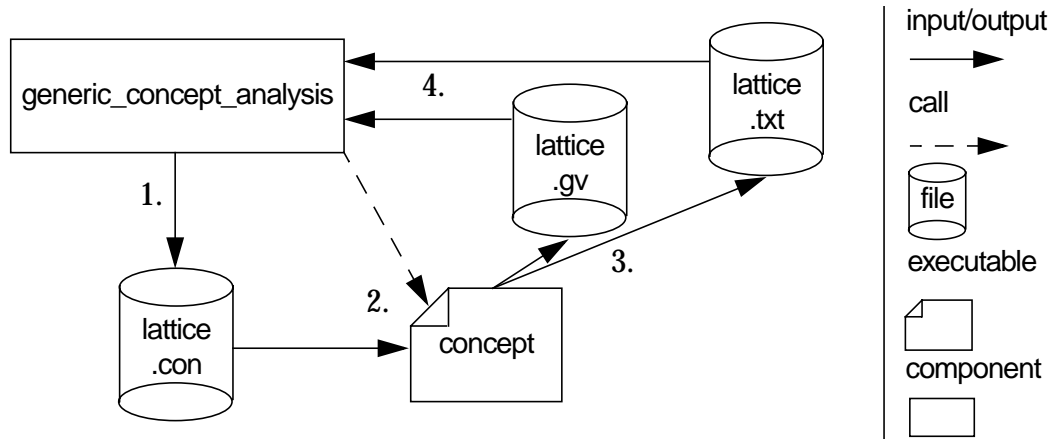


Figure 4-3. **Call of concept and File Handling**

The format of the output files of `concept` can be chosen by the user. It was specified in a way that eases reading. The output file of `generic_concept_analysis` (`lattice.con`) has to have the format requested by `concept`. This format was already explained in Section 2.3. Because of the mapping of objects and attributes to numbers, only numbers are used in the actual format of the input and output files of `concept`. In Figure 2-2, 2-3 and 2-4 the string representation of the objects and attributes is used.

Examples for the output files resulting from a call of `concept` on a very small C-program are given in Figure 4-4 and Figure 4-5. Remember that objects and attributes are mapped to numbers. As one can see, the contents of the `concept`

nodes in the .gv-file shown in Figure 4-5 seem to be unequal to the contents of those in the .txt-file shown in Figure 4-4. The difference is as follows. As described in Section 3.1, there is a notation for a lattice which allows to assign objects or attributes to only one concept node. This notation is based on the functions  $\sigma$  and  $\tau$ . It provides a sparse representation of the concept lattice. The .gv-file contains a sparse representation of the lattice. In the .txt-file the concepts are marked with all of their objects and attributes.

---

```
N0
O4 3 2 1 0
A1
N1
O4
A1 4
N2
O3 2
A1 3
N3
O1
A1 2
N4
O0
A1 0
N5
O
A1 4 3 2 0
```

Figure 4-4. **File lattice.txt: Concept Node Information**

---

Figure 4-4 shows the file that is used to gather information on the concepts. Each concept has a unique number that will be used to specify the subconcept relation. The number is written after an “N”. From the next line on, starting with an “O”, the objects of the concept are listed. They may be printed in several lines. The last information given on the concepts is the attribute list. The letter “A” is followed by the attributes of the concept, which may also be printed in several lines. The capital letters N, O, A are used to find the beginning of a new concept, its object list and its attribute list, respectively. The concept with the number 0 is the bottom concept of the lattice, the concept with the highest number is the top concept of the lattice.

Figure 4-5 shows the file resulting as output for graphplace – a tool used by Christian Lindig and Gregor Snelting to visualize the lattice. The last information contains the subconcept relation, in other words, the edges between con-

cepts in the lattice. The first number is the subconcept. The second number represents its superconcept.

The file containing the subconcept relation uses the sparse representation for concept lattices. This representation is explained in Section 3.1. The sparse representation saves place, but does not allow to get the contents of concepts easily.

---

```
%!PS
%% number of objects: 5
%% number of attributes: 5
%% number of concepts: 6

() () (1) 0 node
(4) () (4) 1 node
(32) () (3) 2 node
(1) () (0) 4 node
() () () 5 node

0 1 edge
0 2 edge
0 3 edge
0 4 edge
1 5 edge
2 5 edge
3 5 edge
4 5 edge
```

Figure 4-5. File `lattice.gv`: Information on the Subconcept Relation

---

The information on the resulting lattice is gathered from the two files `lattice.txt` and `lattice.gv`. Afterwards all files used for concept are deleted. While reading the concepts and the subconcept relation from the files, the routines `make_concept`, `insert_concept` and `insert_subconcept_edge` of **concepts** are called to insert the concepts and the edges in the resulting view.

#### 4.4.2 concepts

This is the component that really builds the resulting lattice as a view of the RFG. It is not to be mistaken for `concept` – the tool implemented by Christian Lindig. The component **concepts** uses an instance of **generic\_concept\_analysis**. The routines to fill the relation table of this instance are only handed over from **build\_lattice**. The component fills the table, calls the generic lattice building and creates a new view in the RFG. The concepts and edges are inserted into this view. The resulting view is then returned to **build\_lattice** as result.

```
package Concepts is
  -- datatypes
  -- Objects and Attributes
  subtype Object_Type is Rfgs.Node_Ptr;
  subtype Attribute_Type is Rfgs.Edge_Ptr;

  -- the procedures to fill the relation

  procedure Init_Relation;
  -- Creates a start state for the relation. This procedure has to be called before Objects, Attributes or Relations
  -- can be inserted. Build_Lattice will not work properly, if Init_Relation was not called.

  procedure Add_Object(The_Object : Object_Type);
  -- Adds an object to the relation table.

  procedure Add_Attribute(The_Attribute : Attribute_Type);
  -- Adds an attribute to the relation table.

  procedure Add_Relation(The_Object : Object_Type; The_Attribute : Attribute_Type);
  -- Adds the entry (The_Object,The_Attribute) to the relation. Afterwards The_Object "has" The_Attribute.

  procedure Destroy_Relation;
  -- Frees the memory used by the relation.

  -- the function to build the lattice the lattice-view is returned

  Empty_Table : exception;

  function Build_Lattice (The_Rfg : Rfgs.Rfg; Lattice_View_Name : String) return Rfgs.View;
  -- this function starts the lattice-building process
  -- it should only be called AFTER the relation table is filled !!
  -- raises Empty_Table if the table contains no entries

end Concepts;
```

Figure 4-6. **concepts.ads**

---

The type of objects and attributes is declared in the package specification. These types are used for the instantiation of **generic\_concept\_analysis**. Hence, the package **concepts** depends on the implementation of RFGs.

### 4.4.3 build\_lattice

The interface to the RFG and some of the analyses is **build\_lattice**. It provides the function to build a lattice. The input is an **Object\_Predicate** for the objects and a list of **Edge\_Predicates** for the attributes. The objects and attributes for the context are filtered from the RFG using the **Object\_Predicate** to gather the objects and the **Edge\_Predicates** to detect attributes of the gathered objects. These objects and their attributes are given to **concepts** to fill the relation table. In addition to the

predicates, the `User_View` and the `Environment_View` of the RFG are handed over, so that entities contained in either one or both of these views will not be used by the analysis. The `Base_View` parameter is the original view, delivering the entities to be regarded for atomic component detection. The `Lattice_View_Name` is the name of the new view in `The_Rfg` created by **build\_lattice**.

The function `Transform` may be needed to return the `Edge_Ptr` leading to the `Node_Ptr` which fulfills the `Object_Predicate`. For example, the algorithm of `Siff` and `Reps` is looking for ADTs. One of the attributes detected by this algorithm is a violation of the information hiding principle. If a subprogram uses the internal structure of a global variable it has a relation to this variable marked by an internal access flag. If one is looking for the type of the variable, one has to find the related type of this variable. `Transform` returns – in this case – an edge to the type of the variable accessed by the subprogram.

---

```
package Build_Lattice is

  type Edge_Transformal is access
    function(The_Rfg : Rfgs.Rfg; The_Edge : Rfgs.Edge_Ptr) return Rfgs.Edge_Ptr;

  -- the perform function

  function Perform
    (The_Rfg      : Rfgs.Rfg;
     Base_View    : Rfgs.View;
     User_View    : Rfgs.View;
     Environment_View : Rfgs.View;
     Lattice_View_Name : String;
     Object_Predicate : Rfgs.Predicates.Node_Predicate;
     Edge_Predicates : Rfgs.Predicates.Lists.Edge_Predicate_List;
     Transform     : Edge_Transformal)
    return Rfgs.View;

end Build_Lattice;
```

Figure 4-7. **build\_lattice.ads**

---

The result of **build\_lattice** is the view containing the lattice. The RFG given as parameter will be changed by **build\_lattice**.

---

## 4.5 *Design of the Algorithms*

Each of the four approaches based on concept analysis derives its candidates from the lattice or the table containing the relation between objects and attributes. For some approaches it is sufficient to build the lattice and use the lattice to identify component candidates. Thus a generic algorithm (**generic\_ca**) was implemented. This generic algorithm builds a lattice after filling the relation table. Afterwards the Perform function performs Make\_Analysis on a copy of the lattice. For the copy, a new view in the RFG is created which contains the atomic components after Perform was called. The approach of Michael Siff and Thomas Reps enhances the relation table before building the lattice. Thus, this approach can not use the generic algorithm.

---

```
generic
  -- This procedure will work on a copy of the lattice to analyze the structure of the code.
  with procedure Make_Analysis(The_Rfg : Rfgs.Rfg; The_View : Rfgs.View);

package Generic_Ca is

  -- build_lattice has to be called BEFORE perform is called

  function Build_Lattice
    (The_Rfg      : Rfgs.Rfg;
     Base_View    : Rfgs.View;
     User_View    : Rfgs.View;
     Environment_View : Rfgs.View;
     Object_Predicate : Rfgs.Predicates.Node_Predicate;
     Edge_Predicates : Rfgs.Predicates.Lists.Edge_Predicate_List;
     Lattice_Name   : String;
     Transform      : Standard.Build_Lattice.Edge_Transformal)
    -- Transform: This function returns an edge to the kind of node which is inspected for the analysis.
    -- e.g.: Subprogram->Variable->Type
    -- if the analysis is looking for ADTs, the input is the edge from Subprogram to Variable, the output the
    -- edge from Variable to Type

    -- the perform function

  function Perform (The_Rfg : Rfgs.Rfg; View_Name : String) return Rfgs.View;

end Generic_Ca;
```

Figure 4-8. **generic\_ca.ads**

---

In Figure 4-9 the structure of the algorithms and the components they are based on is shown. As one can see, the approach of Siff/Reps is directly based on **con-**

**cepts.** The variable usage table is enhanced before the lattice is built. The component **build\_lattice** filters the data from the Base\_View and then builds the relation table immediately. It is not possible to modify the table before building the lattice. Thus, **build\_lattice** does not suffice. The approach of Sahraoui et. al. does not need a copy of the lattice to perform its analysis. Only some concepts are needed. Thus this approach bases on **build\_lattice**.

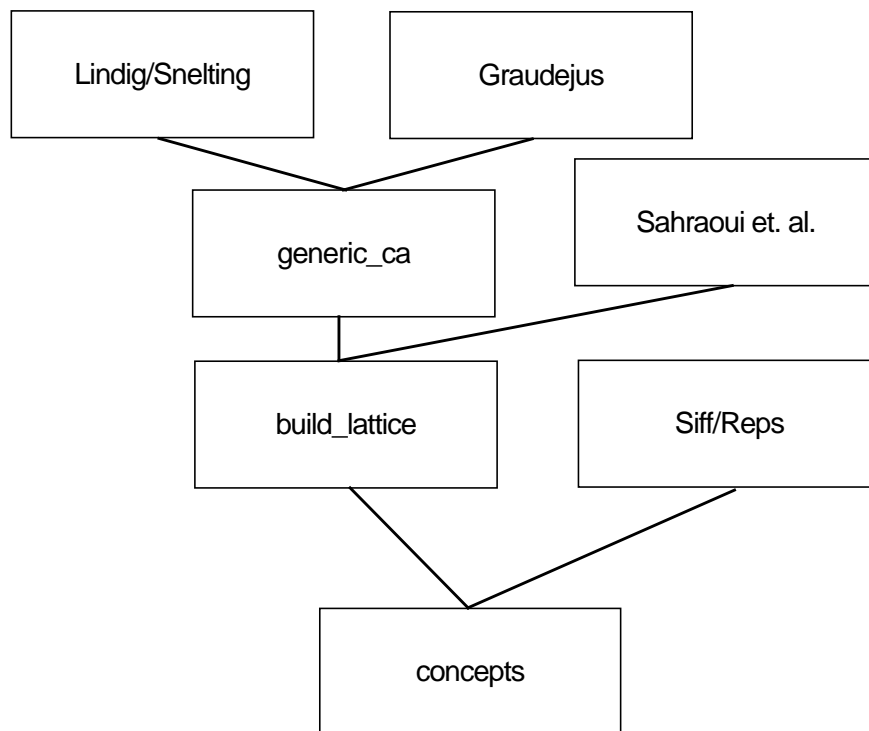


Figure 4-9. **components used by the algorithms**

There is only one function in the package specification of the algorithms. Each of them is exporting the Perform function. The parameters of Perform are The\_Rfg, the Base\_View, the User\_View, the Environment\_View and the AC\_View\_Name. The return parameter is of type Rfgs.View (Figure 4-10).

The Base\_View is – in the context of the algorithms – the view in The\_Rfg containing the input entities and relations for the algorithms. Nodes that are visible in the User\_View or the Environment\_View are not associated with new components. The AC\_View\_Name is the name of the resulting view.

---

```
function Perform
(The_Rfg      : Rfgs.Rfg;
 Base_View   : Rfgs.View;
 User_View   : Rfgs.View;
 Environment_View : Rfgs.View;
 AC_View_Name : String)
return Rfgs.View;
```

Figure 4-10. **the Perform-function of an analysis**

---

---

## **4.6 Incremental Approach**

The User\_view of the RFG contains accepted components. There are at least two ways to use this information. An analysis can ignore elements of these components. Relations between elements of accepted components and analyzed entities will then be lost. Alternatively, the elements of a component can be merged to a node (or the component node is used as representation of its elements) and this node is inspected by the analysis in addition to the other entities. The second approach allows to use the knowledge that some entities represent a component. This knowledge leads to relations of this component to other entities – providing additional information on this component.

In an incremental approach, the components already accepted by the user have an influence on the results of the algorithms based on concept analysis. Objects gathered from the Base\_View already contained in an atomic component are merged to one atomic component entry in the relation table. The attributes related to the objects in the atomic component are gathered from the Base\_View. Afterwards these attributes are united and related to the atomic component entry in the relation table. Hence, let  $o \in O$  be an object contained in the atomic component  $c$ . Then  $\forall a \in A : oRa \Rightarrow cRa$  holds.

For example, if the objects  $o_1$  and  $o_2$  in Figure 4-11 are part of an accepted component, they are merged to one node. The entries in the table are combined using a logical “or”. The result of this operation is shown in Figure 4-12.

The result of an analysis may now provide additional information on the component  $AC_1$ . This may increase the understanding of the analyzed system. The information on the system is refined, considering the actual level of knowledge.

	$a_1$	$a_2$	$a_3$	$a_4$
$o_1$	-	X	-	X
$o_2$	-	X	X	-
$o_3$	X	-	X	-
$o_4$	-	X	X	X

Figure 4-11. **Example Relation Table**

---

	$a_1$	$a_2$	$a_3$	$a_4$
$AC_1$	-	X	X	X
$o_3$	X	-	X	-
$o_4$	-	X	X	X

Figure 4-12. **Example Relation Table with Merged Node**

---



- Reuse
- Lattice Building
- Approaches

---

In this chapter, the implementation of the approaches is described shortly. Moreover, the interface to the tool by Christian Lindig – concept – is presented.

---

### ***5.1 Reuse***

Bauhaus has a variety of code to reuse. Data types, like sets, lists and some other program units, are stored in the directory `/bauhaus/Ada-Code/Reuse/src`. Further reusable units can be found at `/usr/local/lib/reuse`.

In the implementation of the lattice building and the algorithms, some of these components were used.

---

### ***5.2 Lattice Building***

The process of building the lattice representing a given context is implemented using the tool `concept` by Christian Lindig. This tool works with files as input and output. Thus, a program unit was implemented to encapsulate the call of the tool. This component – **`generic_concept_analysis`** – also deals with the files. The call of the tool is handled by subprograms provided by the `gnat` library. In Figure 5-1 one can see how the call is handled.

For file input and output, the units `Ada.Text_IO` and `lo` are used. `lo` can be found in the directory for reuse (`/usr/local/lib/reuse`). The format of the input and output is ASCII. The numbers in the relation table of **`generic_concept_analysis`** are converted to strings. This adds a leading space character to each number.

The program unit **`concepts`** instantiates **`generic_concept_analysis`**.

```
procedure Compute_Lattice is
  Ok : Boolean;
  Argumentlist_Nodes : GNAT.OS_LIB.Argument_List(1..4);
  Argumentlist_Edges : GNAT.OS_LIB.Argument_List(1..3);
begin
  Argumentlist_Nodes(1):=new String("-olattice.txt");
  Argumentlist_Nodes(2):=new String("-c");
  Argumentlist_Nodes(3):=new String("-fN%i%nO%O%nA%A%n");
  Argumentlist_Nodes(4):=new String("lattice.con");
  Argumentlist_Edges(1):=new String("-olattice.gv");
  Argumentlist_Edges(2):=new String("-G");
  Argumentlist_Edges(3):=new String("lattice.con");
  GNAT.OS_LIB.Spawn("./concepts", Argumentlist_Nodes, Ok);
  GNAT.OS_LIB.Spawn("./concepts", Argumentlist_Edges, Ok);
end Compute_Lattice;
```

Figure 5-1. **The Call of concept**

---

---

## 5.3 Algorithms

In the following, the implementation of the algorithms is described. Most of the approaches were not fully described in Chapter 3 and Chapter 4. In the next sections, each algorithm will be presented in more detail. The analysis process is subdivided into logical steps and each of these steps is inspected.

### 5.3.1 Lindig and Snelting

There are mainly two phases of the algorithm. The first phase is the collapse phase. In this phase, the nodes belonging to a horizontal summand are collapsed to one node. Interferences are not collapsed. The second phase is detecting the interferences and resolves them. Afterwards the remaining nodes – excluding the top and bottom node – represent the horizontal summands and thus are the component candidates.

#### 5.3.1.1 Collapse Phase

The algorithm of Lindig and Snelting uses a variant of the biconnected component search to detect interferences. Additionally  $k$ -connected components are inspected to detect all possible interferences. An algorithm to detect the maximal  $k$  of  $k$ -connected components does exist. However, the algorithm does not return the components themselves. To return the components, the algorithm would have needed modification. Because the time needed for these modifications was not accounted for in my schedule, I decided to try an alternative

approach (sharing the same general idea of detecting horizontal summands and resolving the interferences).

In the collapse phase all nodes definitely belonging to a horizontal summand are collapsed to one node representing this horizontal summand. Their intents and extents are merged. If there is an interference, the interfering node will not be merged with the others. Thus the interfering node will still be interfering after the collapse phase. Consider Figure 5-2 as an example.

In Figure 5-2 one node from each horizontal summand participates in the interference. In *horizontal summand 1* the node has a subconcept. Subconcepts are collapsed, too, even if they are below the level of the interference. The result of the collapsing is a directed graph which contains only candidates for horizontal summands and interference nodes. To get the horizontal summands, the interferences have to be resolved.

### 5.3.1.2 Interference Resolution

The concepts are inspected top-down in the lattice. If a node has more than one successor, an interference is detected. The best successor for the interfering variables is computed and the interfering node is merged with the node representing the horizontal summand. The edges to other successors are removed from the graph. The overlapping variables are deleted from the horizontal summands of the loosing successors. The connection to the bottom node may be broken by resolving an interference. In Figure 5-2 an example is given for a single interference resolution. The edge to the horizontal summand on the right is deleted. The interfering node is merged with the horizontal summand on the left.

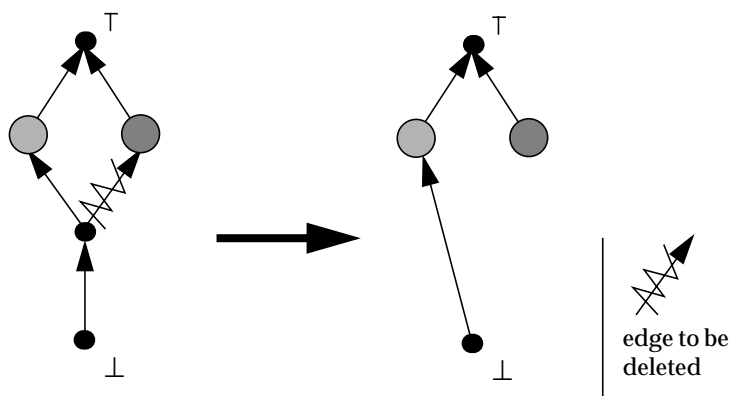


Figure 5-2. A Single Interference Resolution

The component candidates are the nodes representing the horizontal summands. After the interference resolution, only the top and bottom node and the component candidates are left in the graph.

### 5.3.2 Sahraoui et al.

The approach of Sahraoui has three phases. First, the DO candidates are collected. Afterwards overlaps between these candidates are resolved. The results are the detected DOs containing the variables. At last, the subprograms are assigned to the DOs as their methods.

#### 5.3.2.1 Collection of DO Candidates

The search for DO candidates is a top-down search. The first possible candidates are the concepts with the highest number of subprograms. Detected candidates are copied to the result view of the algorithm. The property of the lattice – the partial order – is maintained. Edges may have to be inserted. See Figure 5-3 for example.

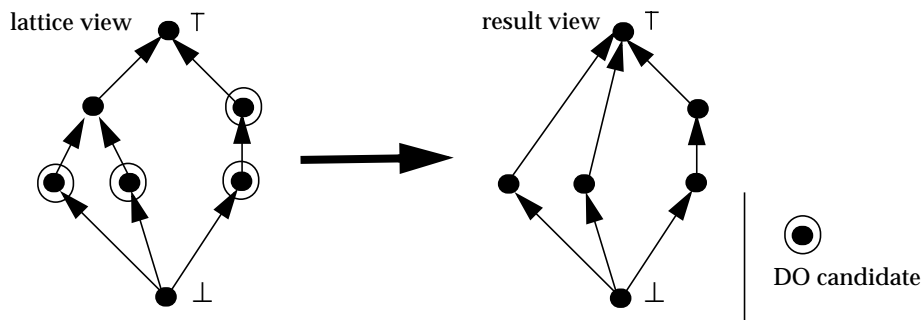


Figure 5-3. Copy of DO Candidates

---

#### 5.3.2.2 Overlap Resolution

In the result view there may be overlaps between DO candidates. In my implementation, all concepts overlapping in at least 50% each are merged.

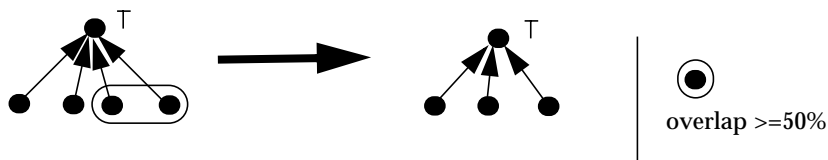


Figure 5-4. Copy of DO Candidates

---

### 5.3.2.3 Method Assignment

The methods are gathered. The referenced and modified variables are inspected for each of the methods. If some variables are modified by the method, the method is deleted from the DOs containing only referenced variables.

### 5.3.3 Siff and Reps

This approach calculates various proposals for components. To have only one result, an evaluation function was implemented. The result is a float number. The higher the number, the better the quality. This function is inspecting the internal access of subprograms to user-defined data types. For each concept the relations between subprograms and types inside and outside of the extent of the concept are checked. If a subprogram has an internal access to a type inside the extent, the result value is increased by 0.1. If there is an internal access to a type outside of the extent, the result value is decreased by 0.2.

---

```

procedure Internal_Access_Check(The_Edge : Rfgs.Edge_Ptr) is
  Transformed_Edge  : Rfgs.Edge_Ptr := Transform(The_Rfg, The_Views, The_Edge);
  The_Node          : Rfgs.Node_Ptr;
begin
  if not Rfgs."="(Transformed_Edge, null) then
    The_Node:=Rfgs.Accessors.Target(The_Rfg, The_Edge);
    if Node_Ptrs.Sets.Is_Member(The_Intent, The_Node) then
      Result:=Result+0.1;
    else
      Result:=Result-0.2;
    end if;
  end if;
end Internal_Access_Check;

```

Figure 5-5. **Internal Access Check**

---

### 5.3.4 Graudejus

The main part of the approach of Graudejus is the contest of overlapping ceiling concepts after the lattice was built. As long as there are overlapping ceiling concepts, the overlap is removed from the looser and the looser is inserted in the lattice. The process of resolving the overlaps is shown in Figure 5-6.

When no overlap exists, the process terminates. The ceiling concepts are the component candidates.

---

```
while not Node_Ptr_Lists.IsEmpty(Ceiling_List) loop
  Overlap:=False;
  First_Node:=Node_Ptr_Lists.FirstValue(Ceiling_List);
  List_Iter:=Node_Ptr_Lists.MakeListIter(Ceiling_List);
  Node_Ptr_Lists.Forward(List_Iter);
  loop
    exit when not Node_Ptr_Lists.More(List_Iter);
    Node_Ptr_Lists.Next(List_Iter, Compare_Node);
    Overlap:=Is_Overlapping_Pair(First_Node, Compare_Node);
    exit when Overlap;
  end loop;
  if Overlap then
    Resolve_Overlap(First_Node, Compare_Node);
  else
    Node_Ptr_Lists.DeleteItems(Ceiling_List, First_Node);
    -- if no overlap occurred the concept can be ignored
  end if;
end loop;
```

Figure 5-6. **Overlap Resolution Loop**

---

- Bauhaus Benchmarks
- References
- Comparison with other Algorithms

---

In the following, a framework for performing benchmarks on clustering techniques will be described briefly. Terminology for the benchmark follows. Then the results of the concept-analysis-based component recovery techniques I implemented applied to the benchmarks will be presented and discussed. In the conclusion, the efficiency and quality of the algorithms will be described.

---

### **6.1 *Bauhaus Benchmarks***

An automatic approach on detecting components in a software system has to be evaluated. A method to rate such an approach is to compare the results with a trusted reference corpus. The reference corpus can be created by maintenance engineers in a manual or semi-automatic way.

In order to compare different approaches, one has to use a common pool of software systems to analyze and a standardized evaluation method. In [KosEis00] a framework is described which provides such a method. Four software systems are analyzed in this section: *Aero*, *Bash*, *CVS*, and *Mosaic*. References for these systems are provided to allow comparison with the results of other algorithms.

In Figure 6-1 the analyzed systems are presented. Their name, their version, the lines of code, the number of user-defined types and global variables and the number of subprograms may give an overview on these systems. The lines of code are commented lines of code. The number of global variables includes constants. *Aero* is an X-Window-based simulator for rigid body systems, *Bash* is a Unix shell, *CVS* is a tool for controlling concurrent software development and *Mosaic* is an internet browser. The graphical user interface of *Mosaic* is not included in the analysis.

System Name	Version	Lines of Code	#User Types	#Global Variables	#Subprograms
Aero	1.7	31.000	57	480	488
Bash	1.14.4	38.000	60	487	1002
CVS	1.8.1	30.000	41	386	575
Mosaic	2.7b	37.000	79	269	564

Figure 6-1. Suite of Analyzed C Systems

The references for these systems were created by software engineers. A detailed documentation on the reference creation can be found in [GirKos99] or at [references]. The number of atomic components found by the software engineers is presented in Figure 6-2.

System Name	#ADT	#ADO	#Hybrid	#Others	#Total
Aero	9	16	1	-	26
Bash	18	16	5	-	39
CVS	13	35	6	-	54
Mosaic	11	29	13	13	66

Figure 6-2. Number of Atomic Components in Analyzed Systems

Mosaic is the only system that contains other components than ADTs, ADOs, or hybrids. A **hybrid** is a component containing subprograms, types, and global variables. The number of *reference components* of the analyzed systems should be compared with the *false positives* and the *true negatives* in the comparison tables, as defined in the following section.

## 6.2 Terminology of the Benchmark

The components detected by an automatic technique are called **component candidates**. The components in the reference corpus are called **reference components**. The evaluation of the result given by an algorithm compares the component candidates (**candidates**) with the reference components (**references**). There are four ways in which a component candidate  $C$  and a reference component  $R$  may be related:

- $C$  and  $R$  overlap in large parts each (**1:1**),
- $C$  is a subset of  $R$  to a large extent (**n:1** –  $C$  is too detailed),

- $R$  is a subset of  $C$  to a large extent (**1:n** –  $C$  is too large) or
- $C$  and  $R$  are not (sufficiently) related.

The best result would be that each component candidate  $C$  would be equal to a reference component  $R$ . However, pragmatically we also consider two components a match if they overlap to at least 70%. If there is no real match between two components, one could still be a subcomponent of the other one (1:n or n:1 match). A component,  $A$ , is a subcomponent of a component,  $B$ , if at least 70% of  $A$ 's elements are also in  $B$ .

In the presence of these imprecise matches, an accuracy factor has been associated with each match that measures the quality of a relation between a candidate and a reference. The **accuracy** between a candidate  $C$  and a reference  $R$  is computed as follows:

$$accuracy(C, R) = \frac{|C \cap R|}{|C \cup R|}, \text{ where } C \text{ and } R \text{ stand for a set of entities.}$$

For matches between more than two components (1:n and n:1) the union of all  $n$  components is used for the accuracy.

If there are some component candidates with no sufficient relation to the reference components, they are called **false positives**. These are candidates which are not represented in the reference and, thus, are false candidates. If there are some reference components with no sufficient relation to the component candidates, they are called **true negatives**. The existence of true negatives leads to unmatched references. The best evaluation result would contain no false positives and no true negatives.

To quantify the detection quality of the algorithm, the **recall rate** is computed. The recall rate abstracts from the level of granularity – since all positive relations are treated equally – and ignores the number of false positives. Thus, the false positives have to be given extra inspection. The recall rate is computed as follows:

$$\text{recall rate} = \frac{\sum_{c \in 1:1} accuracy(c) + \sum_{c \in 1:n} accuracy(c) + \sum_{c \in n:1} accuracy(c)}{|1:1| + |1:n| + |n:1| + |\text{true negatives}|} .$$

In the equation,  $|x|$  denotes the number of elements in the set  $x$ .  $|1:1|$  denotes the number of 1:1 relations, for example. In the following section, the evaluation of the algorithms is discussed.

---

## **6.3 Evaluation**

Each of the algorithms analyses the systems Aero, Bash, CVS, and Mosaic. The result is compared to the reference corpus and the quality of the concept analysis algorithms is compared to other algorithms, including more traditional techniques, namely, *Same Module*, *Internal Access*, *Part Type* and *Global Reference*. For a description of these approaches see [GirKos99], for example.

The reference components contain all components identified by the software engineers: ADOs, ADTs, hybrids and other components. The algorithms are proposing either ADOs or ADTs. Consequently, the result of the algorithms is evaluated with a view containing only the type of components detected by the algorithm. This view also includes all hybrid components because techniques for ADO or ADT recovery may at least identify hybrids partially.

Different aspects are considered to allow a detailed survey of the results. The recall rate and the number of false positives would suffice to give an overview of the quality of the techniques. However, a simple example shows that those two numbers alone might mislead the reader: If a technique proposes one single candidate that is actually the union of all references, the recall rate is 100% and there are no false positives. Nevertheless, a better result would match each reference separately.

The algorithm of Siff and Reps is the only algorithm which enhances the context by adding complement attributes. Thus, many more concepts are created. The analysis of the algorithms was tried on different workstations, but it was impossible for me to get a result. The memory configurations of the machines were:

- *goethe*: **memory**: 320MB **swap**: 656MB
- *zweig*: **memory**: 1024MB **swap**: 359 MB.

The tool of Christian Lindig to build the lattice died with a memory error (malloc failed). Apparently, the number of concepts resulting from enhancing the relation table with complement attributes is too large. The number of concepts seemed to be too large to be computed by concept on these workstations. Thus, there are no results of this approach to be compared.

### **6.3.1 Recall Rate**

In this section, only the recall rate is given thought. Because two of the algorithms detect ADTs and the other two detect ADOs, two diagrams are shown which contain only the relevant techniques. To compare the concept analysis approaches, the recall rates of other approaches are given, too.

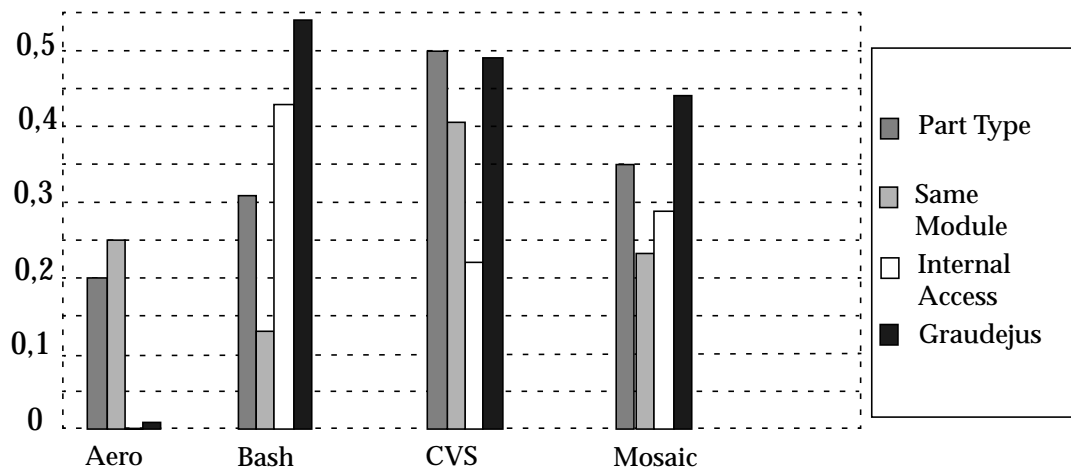


Figure 6-3. Recall Rate for ADTs

The recall rate of the algorithm of Graudejus varies strongly with the kind of system that is analyzed. The approach of Graudejus is the best approach considering only the recall rate.

The approaches of Sahraoui and Lindig/Snelting have leading recall rates. Only once the recall rate of *Same Module* is higher than the one of the concept analysis approaches.

### 6.3.2 Measured Data

A more detailed comparison will now be given. Other approaches and the approaches using concept analysis were evaluated and the results will be presented. The relations are classified into *1:1*, *n:1* and *1:n* relations with an *accuracy* for each type of relation. The number of *candidates*, the *false positives*, and the *true negatives* will be given.

The number of *true negatives* in the comparison tables is the number of ADT or ADO components and hybrids which are not (sufficiently) related to component candidates. In Figure 6-5 the *true negatives* refer to ADTs and hybrids. In Figure 6-6 the number refers to ADOs and hybrids.

The numbers of the reference components of the analyzed systems are shown in Figure 6-2. These values should be compared with the number of *false positives* and *true negatives* in the tables.

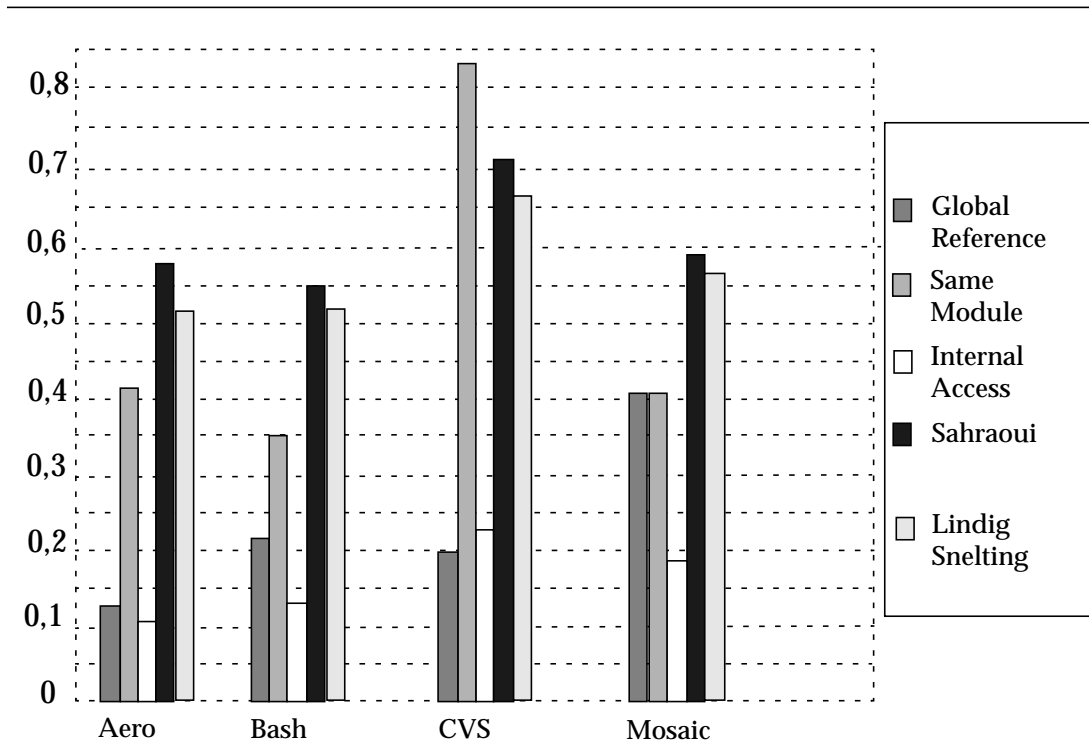


Figure 6-4. Recall Rate for ADOs

The approach of Graudejus produces noticeably more *candidates* than the other algorithms. This leads to a relatively large number of *false positives*. The number and *accuracy* of the relations does not differ remarkably from the other approaches. The approach of Graudejus admittedly has the least *true negatives*. Most of the ADTs and hybrids are detected by the approach of Graudejus with the exception of the system Aero.

The approaches of Sahraoui and Lindig/Snelting produce a huge number of component *candidates*. This leads – like the approach of Graudejus – to a large number of *false positives*. Both approaches have nearly as much *1:1* matches as *Same Module* with a similar *accuracy*. There are much more *1:n* and *n:1* relations with an *accuracy* of about 40%. The number of *true negatives* is very small. Almost all references are covered by the component candidates of the concept analysis approaches.

### 6.3.3 Conclusions

The recall rates of the approaches using concept analysis are the best of the compared approaches. Unfortunately this value alone is not sufficient to qualify for a good component detection algorithm. The *accuracy* of the component candidates is among the average for all evaluated concept analysis approaches. The

Method	System	Candi- dates	1:1		n:1		1:n		False Posi- tives	True Nega- tives
			#	acc %	#	acc %	#	acc %		
Internal Access	Aero	0	-	-	-	-	-	-	-	10
	Bash	18	9	80	3	36	2	59	4	8
	CVS	12	3	82	2	29	4	30	3	10
	Mosaic	17	5	85	2	50	4	41	6	13
Same Module	Aero	5	3	75	-	-	1	30	1	6
	Bash	8	3	81	-	-	1	67	4	19
	CVS	14	5	84	1	29	5	49	3	6
	Mosaic	12	4	89	-	-	6	32	2	14
Part Type	Aero	4	2	77	-	-	1	50	1	7
	Bash	16	7	80	1	44	2	58	6	13
	CVS	21	5	91	4	38	5	31	7	3
	Mosaic	19	6	91	2	68	4	31	7	11
Graudejus	Aero	33	1	75	-	-	-	-	32	9
	Bash	48	7	79	6	53	8	53	27	3
	CVS	48	6	87	3	37	10	43	29	3
	Mosaic	38	4	91	5	54	8	53	17	7

Figure 6-5. ADT Comparison

number of true negatives is average to good considering the approach of Graudejus. Sahraoui and Lindig/Snelting detect nearly all referenced ADOs and hybrids. The worst aspect of the implemented approaches is the aspect of the false positives. All of the approaches produce a large number of false positives. This means, that a user of these approaches has to inspect a large number of component candidates in order to separate the good candidates from the bad ones. The good recall rates may be related to the large number of component candidates – if there are many candidates, there are probably some good ones, too.

The approach of Graudejus seems to be inferior to the compared ones because of the large number of false positives and only average accuracy values. If the user has enough time to deal with the false positives, the low number of the true negatives is a good aspect of this approach.

The approaches of Lindig and Snelting have the weakness of presenting too many component candidates to the user. All other measured data were average or good. The best aspect of these approaches is the low number of true negatives. If there were a mechanism to decrease the number of false positives with-

Method	System	Candi- dates	1:1		n:1		1:n		False Posi- tives	True Neg- atives
			#	acc %	#	acc %	#	acc %		
Internal Access	Aero	17	-	-	4	28	2	25	11	9
	Bash	18	2	92	1	25	2	57	13	17
	CVS	25	6	91	-	-	13	35	6	24
	Mosaic	18	2	88	1	67	14	46	1	27
Same Module	Aero	28	7	88	3	1	1	27	17	5
	Bash	35	5	88	4	32	1	64	25	8
	CVS	52	24	94	4	68	14	68	10	1
	Mosaic	23	13	90	5	46	4	30	1	15
Global Reference	Aero	10	3	91	-	-	-	-	7	14
	Bash	21	5	92	-	-	-	-	16	16
	CVS	17	5	100	-	-	8	48	4	29
	Mosaic	24	12	89	5	46	4	29	3	14
Sahraoui	Aero	234	9	92	14	35	11	62	198	0
	Bash	137	9	84	6	40	10	54	112	3
	CVS	192	12	97	9	44	93	71	78	0
	Mosaic	73	15	84	3	53	43	53	12	4
Lindig Snelting	Aero	121	9	83	12	29	3	39	97	0
	Bash	113	6	88	9	37	8	44	90	1
	CVS	141	14	93	12	32	66	69	49	0
	Mosaic	57	15	87	6	48	26	47	10	4

Figure 6-6. ADO Comparison

out reducing the matching candidates, I would not hesitate to recommend these approaches.

At the moment the approach of Lindig/Snelting seems to be the best. This approach has a low number of candidates – compared with the approach of Sahraoui – and has a very low number of true negatives.

- Conclusions
- Future Research

---

This chapter includes the conclusions of the work considering the concept analysis approaches. In Section 7.2, some ideas are presented which seem to be worthy to be explored.

---

### **7.1 Conclusions**

Using concept analysis to identify components in legacy code is an interesting approach. The mathematical theory is very abstract but may be of use as the results show. Another approach using concept analysis is [Cze00], for example. The technique is used to analyze the tool *xfig* – a graphical editor in the *X Window toolkit*.

The relation between operations in the concept lattice and the structure of the legacy system should be investigated further. Adding complement attributes to the relation table does not seem to be directly related to component detection, for example. If we know more about the relationship between refining the knowledge on components and the corresponding operations in the lattice, better techniques might arise.

The fact that some of the approaches using concept analysis detected nearly all reference components with a very good recall rate shows that concept analysis might be a good supplement for the other Bauhaus component detection approaches. The implemented algorithms run fast enough to be used. They need less than half a minute. A problem to be solved is the number of wrong candidates proposed by the approaches.

The tool concept should be called only once. This will be implemented in November.

---

## **7.2 Future Research**

The results of the analyses need to be inspected in more detail. It is to be proved that the false positives presented by the algorithms are indeed false positives. There are always different points of view on grouping entities to components. Maybe some of the components are good candidates based on a different grouping approach.

The number of false positives has to be decreased. Even if some of them are good candidates, there are way too much to be inspected by a human engineer. Aspects that the false positives have in common are to be detected. This may allow to eliminate false positives.

The distinctive contribution of concept analysis techniques has to be explored. Components detected by concept analysis approaches may be totally different from components proposed by more traditional techniques and, thus, complement these.

The approach of Siff and Reps produces a huge number of concepts. The memory of the workstations did not suffice to test the algorithm. The large number of concepts results from the complement attributes. The number of complement attributes might be decreased as follows. Siff and Reps look for rows in the relation table with a subset relation. For each pair, a complement attribute is inserted, iff there is a real subset. If one would create a partial order of rows based on the real subset relation  $\subset$ , a large number of subset relations may be broken up by adding only one complement attribute. This can be achieved by choosing the longest path within the partial order and breaking the subset relation between the lowest entry and the entry above. All transitive relations to elements above the lowest element are broken up by applying this operation.

---

## Appendix A *Bibliography*

- [ada] <http://www.adahome.com/rm95/>
- [Bar98] John Barnes: *Programming in Ada95*. Addison–Wesley, second edition, 1998.
- [bauhaus] <http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/>
- [Bir40] G. Birkhoff: *Lattice Theory*. American Mathematical Society, Providence, R.I., first edition, 1940.
- [Boe81] Barry W. Boehm: *Software Engineering Economics*. Prentice Hall, Upper Saddle River, New Jersey 07458, 1981.
- [Can96] G. Canfora, A. Cimitile and M. Munro: An Improved Algorithm for Identifying Objects in Code. In *Journal of Software Practise and Experience*, Pittsburgh, volume 26, number 1, pages 25–48, January 1996.
- [concepts] <ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/misc/>
- [Cze00] J. Czeranski, T. Eisenbarth, H. Kienle, R. Koschke, D. Simon: Analyzing xfig Using the Bauhaus Tool, In *Working Conference on Reverse Engineering WCRE'00*, pages 23–25, Brisbane, Australia, November 2000. IEEE Computer Society Press.
- [GirKos99] J.–F. Girard and Rainer Koschke: A Comparison of Abstract Data Type and Objects Recovery Techniques. In *Journal Science of Computer Programming*, Elviesier, 1999.
- [Gra98] Holger Graudejus: *Implementing a Concept Analysis Tool for Identifying Abstract Data Types in C Code*. Master thesis, Institute for Computer Science, University of Kaiserslautern, November 1998.
- [Kos00] Rainer Koschke: *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institute for Computer Science, University of Stuttgart, 2000.

- 
- [KosEis00] Rainer Koschke and Thomas Eisenbarth: A Framework for Experimental Evaluation of Clustering Techniques. In *Proceedings of the International Workshop on Program Comprehension, IWPC'2000*, pages 10–11, Limerick, Ireland, June 2000.
- [LinSnel97] Christian Lindig and Gregor Snelting: Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In *ICSE*, pages 349–359, Boston, MA, 1997.
- [LiuWil90] S. S. Liu and N. Wilde: Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery. In *Int. Conf. on Software Maintenance*, pages 266–271, November 1990. IEEE Computer Society Press.
- [LRM94] Ada95 – The Language Reference Manual. International Standard ISO/IEC 8652:1995(E). Version 6.0, 21 December 1994.
- [references] <http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/papers/scp99.html>
- [rigi] <http://www.rigi.csc.uvic.ca/>
- [Sah97] Houari A. Sahraoui, Walcéllo Melo, Hakim Lounis and François Dumont: Applying Concept Formation Methods to Object Identification In Procedural Code. In *Proc. of Conference on Automated Software Engineering*, pages 210–218, November 1997. IEEE Computer Society Press.
- [SifRep99] Michael Siff and Thomas Reps: Identifying Modules via Concept Analysis. In *IEEE Transactions on Software Engineering*, volume 25, number 6, pages 749–768, November/December 1999.
- [Sne95] Gregor Snelting: *Reengineering of Configurations Based on Mathematical Concept Analysis*. Informatik–Bericht Nr. 95–02, Abteilung Softwaretechnologie, Technische Universität Braunschweig, Januar 1995.
- [Yeh95] A. S. Yeh, D. Harris and H. Reubenstein: Recovering Abstract Data Types and Object Instances From a Conventional Procedural Language. In *Second Working Conference on Software Engineering*, pages 227–236, July 1995. IEEE Computer Society Press.

## **Erklärung**

Ich versichere, daß ich diese Arbeit selbständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe.

---