Interface Regroup Wizard

Regrouping the symbols of interfaces

Rahmat Adnan

Interface Regroup Wizard

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Rahmat Adnan
born in Surabaya, Indonesia

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

ASML Netherlands B.V.
De Run 6501
Veldhoven, the Netherlands
www.asml.nl

Cover picture: ASML TwinScan.

Interface Regroup Wizard

Author: Rahmat Adnan
Student id: 1015583
Email: r.adnan@gmail.com

Abstract

With every generation, the complexity of the lithography systems produced by ASML increases. This increase of complexity affects the corresponding software (control) systems, i.e., they have become immense and complicated. The increase of complexity also affects the interfaces. An interface is an essential element that allows components in a software system to communicate with each other. As a consequence, a modification of an interface definition has become an expensive operation. In this thesis, we present a tool called the Interface Regroup Wizard (IRW) that is able to assist the architect in regrouping the interfaces by means of hierarchical clustering. We have successfully applied the IRW to the interfaces of the software system developed by ASML. The interfaces are regrouped into smaller interfaces and validated such that a set of criteria defining a correct interface is satisfied.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Ir. B. Graaf, Faculty EEMCS, TU Delft
Company supervisor:	Ir. J. Zonneveld, ASML Netherlands B.V.
Committee Member:	Dr. L. Moonen, Faculty EEMCS, TU Delft
Committee Member:	Ir. B.R. Sodoyer, Faculty EEMCS, TU Delft

Preface

This thesis is based upon studies performed in the period of May to September 2007 at ASML B.V. and Delft University of Technology, The Netherlands.

I would like to express my sincere gratitude to my company supervisors Joost Zonneveld and Remco van Engelen. Without their advice, patience and support this thesis would not have existed. Furthermore, I would like to thank all the architects at ASML for their great cooperation in providing feedback and advice.

I would like to thank Bas Graaf for all his assistance and support. I would also like to thank Arie van Deursen for making this assignment available and for the introduction to ASML. Additionally, I would like to thank Matteucci Matteo of Politecnico di Milano for his help with the hierarchical algorithm.

Finally, I would like to give my greatest appreciation to my family and friends, who have supported me, especially to Ifa Chaeron for her kind love and understanding, which make everything more pleasant and my mother, Dewi Suhufi, for her neverending support.

Rahmat Adnan
Delft, the Netherlands
October 31, 2007

Contents

List of Figures

Chapter 1

Introduction

This thesis describes a research project that was performed to automate the optimization of interfaces. The discussed project was carried out within the context of the Interface Management and Archive Structure (IMAS) project, which is an internal project at ASML. The aim of the IMAS project is to increase the awareness and decrease the impact of interface changes, to obtain a measurable positive impact on the build performance and to have the software archive adhere to the directive set forth by the ASML configuration management project.

This chapter presents the problem definition, which gave birth to this research project, in Section 1.1. We present a brief description of the solution in Section 1.2. In Section 1.3, a discussion is provided about the relevancy of this research project for software engineering in general and software maintenance in particular. Finally, an outline of the thesis presented in Section 1.4.

1.1 Problem definition

ASML is the world's leading provider of lithography systems for the semiconductor industry. The main product manufactured by ASML is the TwinScan lithography system, which we will refer to as TwinScan for convenience reason. A TwinScan is able to print integrated circuits on a thin slice of semiconducting material or a wafer. A TwinScan is often referred to as a wafer scanner due to its scanner-like movements performed on the wafer. A TwinScan is decomposed into smaller hardware blocks, each having its own software system. This software systems consist of components that communicate with each other through interfaces. At ASML, an interface is a collection of *symbols* that can be used by *users* in the components. Here, a symbol represents a macro definition, a type definition or a function declaration. A user is as an implementation file, e.g., `.c` file. Despite that the software system of the TwinScan is written in C, interfaces written in other programming languages consist of more or less the same representations. For our research project, however, we restrict ourselves to the C language.

With every generation, the complexity of producing integrated circuits with more functionality increases, which drives ASML to produce lithography systems that are able to cope

with this increasing complexity. Due to the increase of complexity and the implementation of new technologies, the architecture of the TwinScan machines and their corresponding software systems have become immense and complicated. Currently, a software system for the TwinScan machines contains of approximately 20 MLOC decomposed into thousands of components and interfaces. A software system of this magnitude becomes hard to maintain and a slight modification to the software system, which causes a recompilation, has become an expensive operation. Moreover, the interfaces in the TwinScan software system are:

- large and unbalanced,
- not functional coherent, i.e., the symbols are not functionally related, and
- not encapsulated on a certain level, e.g., functional cluster or building block.

In order to tackle the problems stated above, ASML has set up a set of IMAS compliancy criteria, which all interfaces in the software repository eventually have to comply with. Using the IMAS compliancy criteria, ASML attempts to increase the coherency within an interface and to reduce the build time when a modification to an interface is made. The requirements for an interface set by IMAS are as follows:

- Interfaces shall publish a coherent set of functionalities.
- Interfaces shall be designed for minimum change.
- Interfaces shall use a minimum amount of other interfaces.
- Interfaces shall publish a minimum amount of symbols that are not needed by the clients/users on its level (encapsulation).

Currently, in order to fulfill the IMAS compliancy criteria, the symbols defined in an interface are being regrouped manually. This manual regrouping process is a tedious and error-prone process.

1.2 Solution

Instead of manual regrouping, ASML wishes for a more automated approach that is able to decouple an interface into smaller interfaces based on the functional coherency of the symbols. Closely related to ASML's wishes, It is our task to investigate the possibilities of an either fully automated or human assisted approach, which groups symbols together and generates interfaces that are correct or IMAS compliant.

We have developed a tool called Interface Regroup Wizard (IRW) that is able to (semi) automatically regroup symbols in an interface, such that the groupings are IMAS compliant. The IRW is successful when it is able to regroup non IMAS compliant interface into interfaces that does satisfy the IMAS compliancy criteria.

1.3 Applicability

The IMAS compliancy criteria set by ASML are actually general rules, which can be applied in other software systems with a component based architecture. Therefore, the results and findings of this research project are relevant to improve maintainability of a broader class of software systems. It eases the maintenance process by automatically regrouping the interface definitions. Since the optimal configuration is generated by the IRW, modifications of the interface definitions trigger a minimal recompilation time. Much less effort is needed to keep the software system in a good shape.

We have successfully applied the results of this research to the TwinScan software systems developed by ASML. A case study of a non IMAS compliant interface in the TwinScan software system is taken in order to validate the IRW.

1.4 Outline

In the remaining of this thesis, we discuss the IRW in detail. We start with a description of ASML, the TwinScan, its corresponding software architecture, and the current situation in Chapter 2. Subsequently, we describe the purpose of the IRW, its scope and the requirements in Chapter 3. After a full comprehension of the assignment, we investigate the related works. A presentation of these works and the applicable techniques, i.e., formal concept analysis and cluster analysis, is given in Chapter 4. We continue with the discussion of the algorithms used in the IRW in Chapter 5. In Chapter 5 we also discuss the challenges we encountered and the design of the IRW. The graphical user interface of the IRW, the problems we stumbled upon, and the results generated by the IRW are discussed in Chapter 6. Finally, we provide a conclusion in Chapter 7. Here, we also discuss our contributions and the IRW's contributions for ASML. Furthermore, a brief discussion of the future works concerning the IRW is presented in Chapter 7

Chapter 2

About ASML

ASML is the world's leading provider of lithography systems for the semiconductor industry, manufacturing complex machines that are critical to the production of integrated circuits or chips. ASML designs, develops, integrates, markets and services advanced lithography systems used by customers – the major global semiconductor manufacturers – to create chips that power a wide array of electronic, communications and information technology products [1].

In this chapter we provide information about ASML, which is relevant for the IRW. We begin with a brief discussion of ASML's main product in Section 2.1. In Section 2.2 we continue with a discussion of the software architecture used at ASML. We conclude this chapter with a description of the current situation in Section 2.3.

2.1 TwinScan

Throughout the years ASML has produced high quality lithography products. The TwinScan lithography platform is an example of such a lithography product. The TwinScan lithography platform is composed of multiple TwinScan machines. Each of these TwinScan machines is able to print integrated circuits on a wafer, which is a thin slice of semiconducting material. The printing process consists of multiple stages where each stage is also called as a scan due to the scanner-like movements. Therefore, a TwinScan machine is often referred to as a wafer scanner. ASML's TwinScan lithography platform was introduced in 2000 and it has established itself as the de facto standard for the manufacturing of integrated circuits [2]. The dual-stage design, where wafer measurement and imaging is performed in parallel, started a revolution during its introduction. Figure 2.1 illustrates the parallel processing of wafer measurement and wafer exposure. The success of the TwinScan platform is ascribed to its modular design. Each TwinScan tool can be tailored and upgraded as desired. Currently the latest model of the TwinScan series is the XT:1900 Gi for volume production of integrated circuits at 45 nm and below.

Each TwinScan machine is tailored by configuring its software (control) system such that it fulfill the demands of a specific customer. The software system for a TwinScan machine is designed and developed by ASML. Currently, the size of the software system for

Figure 2.1: Dual-stage metrology layout.

the TwinScan platform is around twenty million lines of code (20 MLOC). The architecture of this software system is hierarchical. The next section discusses the mentioned software architecture.

2.2 Architecture

The architecture of the software system for the TwinScan machines is based on the categorization of the hardware components. A possible categorization process of the hardware components is performed at a high level where the TwinScan machine is divided into a few big blocks. A more plausible and manageable approach is to categorize the hardware of TwinScan machine on a lower abstraction level, such that dozens of small blocks are created.

Figure 2.2 illustrates an example of the hardware categorization of the TwinScan machine. Here, the created categories are called hardware blocks. Each hardware block has its own software system called (software) *building block*. These building blocks are organized and grouped together in logical and functional coherent clusters, which are called *functional clusters*. In other words, a functional cluster is a container for n logically associated building blocks. For every functional cluster an architect is appointed. These appointed architects are called FC-architects. During our research project we mainly communicate with the FC-architects in order to acquire requirements and feedback.

Modifications often occurs during the life span of a software system. For example, when a client desires a new feature, this feature will probably be included in the next software release. At ASML, a *release part* contains several building blocks, which are required for the proper functioning of the software release. Therefore, next to a grouping of building

Figure 2.2: A decomposition of TwinScan into building blocks.

blocks based on their logical functionality, i.e., functional clusters, the building blocks can also be grouped together based on their inclusion in a certain software release, i.e., release parts.

Figure 2.3 illustrates the software hierarchy of a single TwinScan machine. Here we can see that a building block can be owned by either a functional cluster and / or a release part. However, as previously stated, a functional cluster and a release part own a building block on a fundamentally different reason. Furthermore in Figure 2.3, a building block may contain multiple components and a component may consist of n interfaces and files. Note that building blocks contain interfaces transitively through its components.

As mentioned earlier, the size of the whole system is approximately 20 MLOC. This huge amount of code for a single system is distributed in: 3 release parts, 23 functional clusters, 165 building blocks, 355 components and 1550 interfaces. These 1550 interfaces define over half a million symbols.

2.2.1 Interface

Since our research project mainly revolves around interfaces, we will discuss the concept of an interface more thoroughly in this subsection. Basically, an interface is a boundary across which two independent entities meet and interact or communicate with each other [7]. An interface is a contract between the communicating entities. At ASML, an interface is a collection of definition files, i.e., `.h` files and ASML specific `.ci` files, which are equivalent to abstract Java classes containing a mixture of method prototypes and implementations. The definition files in an interface contain *symbols*, i.e., macro definitions, type definitions and function declarations that are accessed by *users*. A user may either be a `.c` file, `.ci`

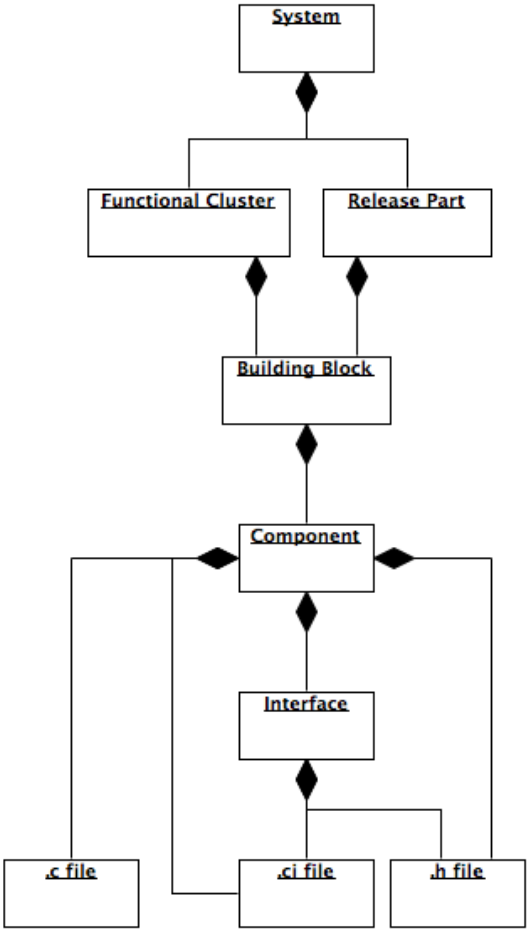

Figure 2.3: The hierarchy of the TwinScan software architecture.

file or `.h` file. A general rule: A symbol is selected as an interface symbol when it has at least one external user, i.e., a user located in another component. So, in other words, an interface is a collection of symbols that can be used externally, in other components.

An interface may be *provided* by building blocks, functional clusters, release parts and the system. The term provided indicates that at least one symbol of an interface has one or more users in other building blocks, functional clusters, release parts or system, respectively. Figure 2.4 illustrates an example of a possible usage of an interface. Interface `I` in functional cluster `A` is used by `bar.c` in functional cluster `B`. As we can clearly see, the arrow specifying the interface usage crosses multiple levels. In this case, we say that interface `I` is provided by functional cluster `A` to be used by users in other functional clusters. If it is not specified, the default level of an interface is component. A component level interface is an interface that may only be used by users of other components in the same building block. In Figure 2.4, Interface `J` is an example of such component level interface where

its user `foo.c` is located in another component `DD`, but in the same building block `XX`. The highest interface provider providing an interface determines the level of the interface. Therefore, the potential level of an interface, from low level to high level, is component, building block, functional cluster, release part or system.

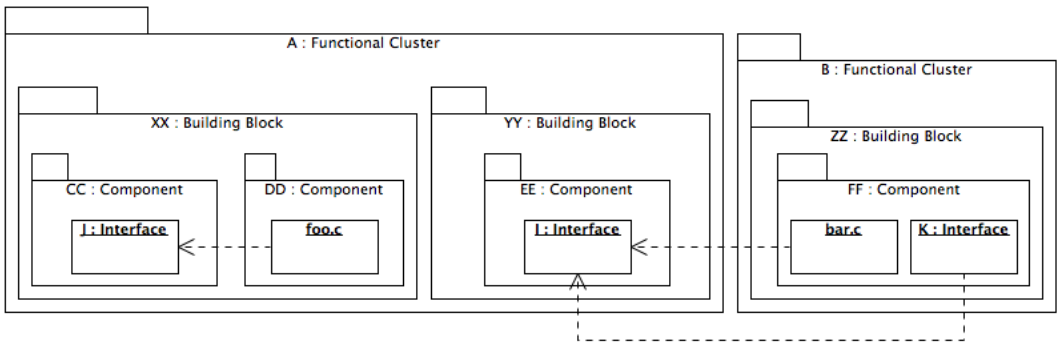

Figure 2.4: Interface usage.

Next to the five levels discussed above, there are three other levels. These levels are indicated by the term *shared*, e.g., release part: Shared, functional cluster: Shared and building block: Shared. A *shared interface* is an interface where at least one symbol is used by other interfaces. Figure 2.4 shows the sharing phenomenon where interface `I` is used by interface `K`. At ASML, sharing is also called as *dealing* because an interface is dealing symbols of other interfaces. For example in Figure 2.4, interface `K` is dealing the symbols of interface `I`. By introducing the shared levels, interface `I` obtains the level of functional cluster: Shared instead of functional cluster.

The level of shared interfaces is different than the five levels discussed earlier. The shared interfaces are located inbetween the non-shared interfaces. For example, an interface that is shared at the functional cluster level is higher than an interface not shared at the functional cluster level, but lower than an interface not shared at release part level. Table 2.1 Illustrates the sequence of levels from high to low level and it also presents the corresponding short description for each level.

2.2.2 IMAS

The interface management and software archive structure (IMAS) is an internal project within ASML. IMAS' main goal is the implementation of an interface change process to streamline and manage interface changes. Furthermore, IMAS has the following subgoals:

- Implement a software and document archive structure in terms of building blocks and functional clusters.
- Obtain a measurable positive impact on the build performance.

Table 2.1: Interface levels from high to low level.

Interface Level	*Description*
System	Interfaces provided to be used by components in other systems.
Release Part: Shared	Interfaces required by at least one interface having release part level.
Release Part	Interface provided to be used by components in other release parts.
Functional Cluster: Shared	Interface required by at least one interface having functional cluster level.
Functional Cluster	Interface provided to be use by components in other functional cluster.
Building Block: Shared	Interface required by at least one interface having building block level.
Building Block	Interface provided to be used by components in other building blocks.
Component	Interface provided to be used by components in the same building block.

According to the IMAS project an interface has to publish a coherent set of functionality, it has to be designed for minimum change, it has to be dealing the minimum amount of other interfaces and it has to publish a minimum amount of symbols that are needed by the clients from other levels (encapsulation). For the purpose of measuring the correctness of interfaces, the IMAS project measures the following properties:

- Change
- Dependency
- Dealing
- Encapsulation

A change is measured as the number of deliveries made for an interface to the repository since January 2005. Change should be minimized.

Dependency is measured as the amount of files that has to be rebuilt when the interface is changed. Dependency should be minimized.

Dealing is the inclusion of other interfaces in an interface. Dealing can be divided into direct and indirect dealing. Direct dealing is the direct inclusion of interfaces. Indirect dealing is the inclusion of interfaces as defined in the interface dependency tree. Figure 2.5 illustrates an interface dependency tree. Here, interface `A` is directly dealing interface `B` and interface `B` is directly dealing interfaces `C` and `D`. Transitively, interface `A` is indirectly

dealing interfaces `C` and `D`. Dealing is measured as the total number of interfaces directly and indirectly dealed. Dealing should be minimized.

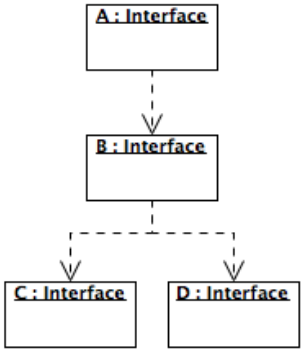

Figure 2.5: An interface dependency tree.

Encapsulation is measured as:

$$\frac{S_{level}}{S_{total}} \times 100\%$$

where S_{level} is the number of exported symbols actually used at the interface's level and S_{total} is the total number of exported symbols. Suppose that an interface having functional cluster level exports 100 symbols and from these 100 symbols, 80 are actually used by users in other functional clusters. Then, the encapsulation measurement returns 80%. Encapsulation should be maximized.

Figure 2.6 illustrates how each measurement results in a score of $0 - 3$. An interface is IMAS compliant when the cumulative score for all measurements is less than or equal to three. Suppose that an interface is critical (score: 3) for the encapsulation criteria, then it is still IMAS compliant when all other criteria are OK (score: 0).

Our research project falls within the boundary of the IMAS project. Therefore, most assistance is provided by the members of the IMAS project. Furthermore, the problem definition, goals and requirements for the IRW are set in agreement with the members of the IMAS project.

2.3 Current Situation

During its lifetime, the TwinScan software system went through many changes. When functionalities are added or removed, a new piece of software is also added or removed. Due to these changes the dependencies have increased. A consequence is that slight modifications of some symbols result in nearly a recompilation of the whole software repository. The number of interfaces that do not meet the criteria set by the IMAS project increases gradually. Currently, 28% of all interfaces, i.e., 470 interfaces, are not IMAS compliant.

points	Average	Ok 0	Medium 1	High 2	Critical 3
Change	2.8	< mean	< 2x mean	< 3x mean	> 3x mean
Dependencies	1260	< mean	< 2x mean	< 3x mean	> 3x mean
Direct dealing	2	< mean	< 2x mean	< 3x mean	> 3x mean
Indirect dealing	11	< mean	< 2x mean	< 3x mean	> 3x mean
Encapsulation		> 80%	60-80%	40-60%	<40%

Figure 2.6: IMAS compliancy threshold.

These interfaces need to be splitted into smaller pieces such that IMAS compliancy criteria are met.

The interface splitting process is performed manually conforming to the interface decoupling guideline set by the IMAS project. The interface decoupling guideline is basically a step-by-step instruction of how to split interfaces. Constraints are set on the guideline, i.e., interface design should never limit machine performance, and interface management should not increase time-to-market. The guideline is based on the general principle of grouping functional coherent symbols in a single interface, and preventing the mix up of symbols used at different architectural levels in a single interface.

An interface is a container for multiple definition files. Each of these files defines the symbols that are exported by an interface. For example interface `AAxBB` consists of the following files:

- `AAxBB.h`
- `AAxBBtyp.h`
- `AAxBBmet.h`
- `AAxBB_diagnostics.h`

Files `AAxBB.h`, `AAxBBtyp.h`, `AAxBBmet.h` and `AAxBB_diagnostics.h` define a set of symbols that is exported by interface `AAxBB`.

Figure 2.7 illustrates a potential conflict concerning the example interface `AAxBB`. In Figure 2.7, `AAxBB`'s level is illustrated as module level. This module level can either be a building block or a higher level. Suppose that `AAxBB` has functional cluster level and the symbols of `AAxBB_diagnostics.h` are only used by users in other building blocks in the same functional cluster. The result of the current grouping is that the symbols of `AAxBB_diagnostics.h` are unnecessarily made available to other functional clusters.

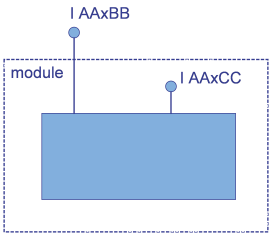

Figure 2.7: A Potential conflict

Figure 2.8 presents a possible solution to the conflict previously discussed. Here, a new interface called `AAxBBDIAG` with building block level is created containing only `AAxBB_diagnostics.h`. By splitting, the symbols of `AAxBB_diagnostics.h` are not made available for users from other functional clusters anymore.

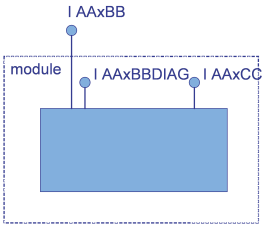

Figure 2.8: A solution for the potential conflict

In the example above, splitting of an interface based on separation of definition files is discussed. In practice however, it is preferred to split an interface at the symbol level, i.e., the type declaration, macro definition and function prototype level. The functional coherent symbols should be grouped together, which forms a new interface. Generally, the interface decoupling guideline proposes the following approach for the split-up:

- Determine which interface needs to be splitted.
- Label each symbol in the interface with the level it should belong to.
- Repeat for each level:
  - Collect all symbols at this level.
  - Evaluate for each symbol whether it should be exported or moved.
  - Group symbols into functional coherent interfaces.
  - Add symbols from lower levels if needed for coherency.
  - Identify all symbols also used in interfaces of other parts/functional clusters/building blocks to find the shared interface symbols.

  - Put the shared interface symbols in separate interfaces.

- Document the new interface(s).

Appendix B offers a detailed step-by-step instructions of how to split an interface as specified by the interface decoupling guideline.

Chapter 3

Requirements Analysis

This chapter describes the results of the requirements elicitation and analysis activities for the development of the IRW. The IRW is developed in the context of the IMAS project at ASML, which aims to optimize the interfaces for the TwinScan software system with respect to various criteria. A detailed description of the IMAS project and its goals have already been provided in Section 2.2.2.

First a general discussion on the purpose of the system is presented in Section 3.1. We continue with the presentation of the functional and the nonfunctional requirements in Section 3.3 and 3.4, respectively. We conclude the chapter with some scenarios and use case models in Section 3.5.

3.1 Purpose

The main objective of the IRW is to reorganize the current interface structure by regrouping the symbols in an interface or multiple interfaces in a component such that the IMAS compliancy criteria are met. The identified goals are:

- Reduce the amount of unnecessary inclusion of symbols.
- Reduce the interface dependencies.
- Increase the grouping of symbols used at the same level.

By reducing the amount of unnecessary inclusion of symbols, we increase the *selectivity*. Some symbols in an interface may not be closely related, but they are still grouped together in a single interface due to certain conventions. Decoupling of these symbols is therefore highly desired.

It is common for an interface to be used by another interface. This is necessary, for instance, when a function prototype in some interface requires a type defined in another interface. This phenomenon, which is also called *dealing*, creates interface dependencies that can be visualized by an interface dependency tree (IDT). Figure 2.5 illustrates such an IDT. Suppose that a user `foo.c` uses interface `A` in Figure 2.5. Intrinsically, this usage results in the usage of all child interfaces of `A` in the IDT, i.e., interfaces `B`, `C` and `D`. The

other way around, the symbols of interface `D` are used by `foo.c` through dependency. Therefore, a modification of one of these symbols will trigger a recompilation of `foo.c`. Consequently, a minimization of interface dependencies is desired and all dealed symbols should be grouped together (encapsulated) in a separate interface.

The concept of *encapsulation* is very important at ASML. Encapsulation prevents a symbol to be provided outside its own usage level. For example, symbols used at a building block level are supposed to be grouped together and they should not be grouped with symbols that are used at functional cluster level. By grouping these building block symbols, we are providing these symbols at the building block level and thus preventing them from being used at higher levels such as functional cluster. However, it is sometimes necessary to group two symbols used at different levels due to functional coherency. A maximization of encapsulation is desired.

The fulfillment of the defined goals have a positive effect on the build time. Another factor that also influences the build time, which we cannot control, is the change rate. The change rate is a dynamic factor that fluctuates depending on the change requests on the ASML machines. When the change rate is stable, the fulfillment of the defined goals results in a reduction of the build time. In other words an increase of the change rate suppresses the benefits obtained by the regrouping of the interfaces.

3.2 Scope

The scope of the system is defined based on the general statement:

> Assist the architect in the decision process of how to regroup the interfaces.

Several questions can be formulated concerning the general statement above, e.g, what should be the outcome/deliverables of the IRW? How does the architect interact with the IRW? What are the required functionalities of the IRW? Where does the input data required by the IRW come from?

The scope of the system can be set by analyzing the context of the system. A system context diagram visualizes the context of a system. It is the highest level view of a system, showing a system as a whole and its inputs and outputs [16]. Figure 3.1 illustrates the system context diagram for the IRW. Corresponding to Figure 3.1, the inputs required and the outputs generated by the IRW are:

- Inputs:
  - Dependency relations between interfaces/symbols and their users.
  - Scope set files.
  - Coherency information provided by the architect.
- Output:
  - Recommendation on the interface splitting provided to the architect for agreement.

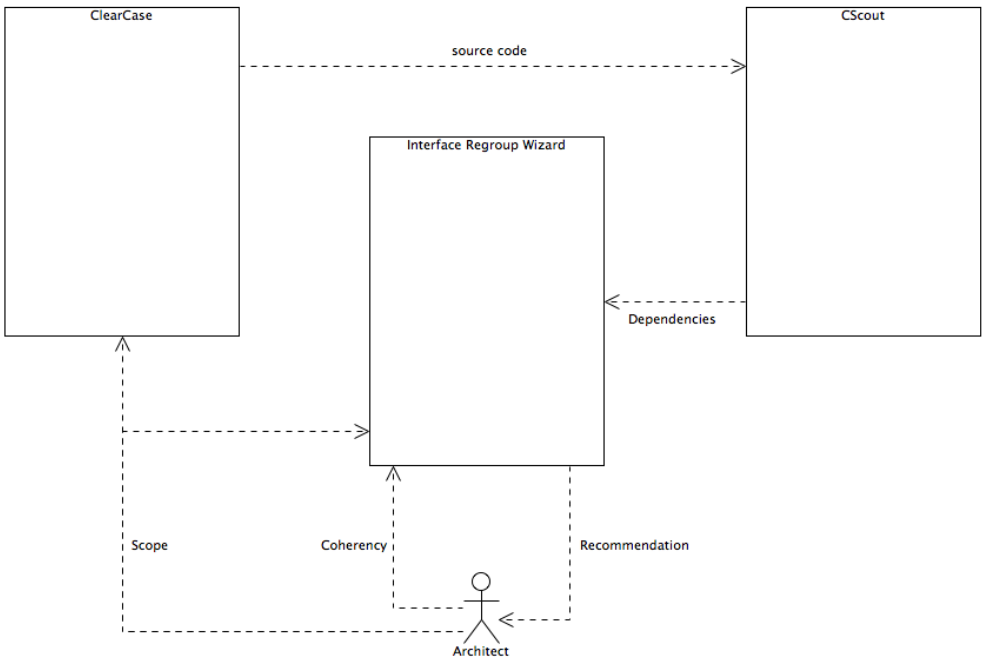

Figure 3.1: System context diagram.

From Figure 3.1 we clearly see that the IRW requires three inputs, i.e., dependency data, scope set files and coherency data. The dependency data is provided by CScout, which is a source code analyzer. A thoroughly description of CScout is provided in Chapter 4. The scope set files on the other hand are created manually by the architects at ASML. The scope set files are text files that describe the boundary of the system and the internal relations of the system components. The structure of the repository in ClearCase, which is a software configuration management system used at ASML, should match the scope set files. The coherency data are provided by the architect during the execution of the IRW or after revising the recommendation generated by the IRW.

3.3 Functional Requirements

Functional requirement is an area of functionality the system must support. The functional requirements describe the interaction between the actors and the system independent of the realization of the system [6]. During requirements elicitation the following functional requirements for the IRW are identified:

1. Collect all symbols of an interface to be regrouped.
2. Group the symbols based on:
    a) user defined grouping.

b) symbol usage level.

c) strongly connected symbols.

d) a number of common clients.

3. Display result.

3.3.1 Collecting symbols

Initially, all symbols of an interface need to be collected in a datafile. This collection of symbols serves as the input data for the IRW.

3.3.2 Grouping methods

User defined. An architect should be able to define which symbols are functional coherent. When a set of symbols are defined as functional coherent, they are grouped together. For example, an architect defines `WHXA_move_wafer` and `WHXA_move_wafer_fcn` as coherent functions. As a consequence, these functions are grouped together.

Usage level. This grouping method attempt to optimize encapsulation, e.g., a symbol used at building block level should not be grouped with a set of symbols used at functional cluster level.

Strongly connected symbols. Symbols in an interface that are strongly connected, i.e., types/functions that depend on each other either directly or transitively, are grouped together. For example in interface `WHXA`, the function `WHXA_wafer_io_to_string` uses a complex type, such as a struct that contains the type `.carrier`. These function and type are grouped together as strongly connected symbols.

Common users. Symbols that are used by more than some *number* of common clients, specified by the architect, are grouped together. For example, assume that for `WHXA` this number is set to 20. Then, if the function `WHXA_wafer_io_to_string` and the enum type `WHXA_lot_control_enum` are used by 25 common clients, these function and type are grouped together.

3.3.3 Present result

The result is displayed as a graph/lattice or a table, which illustrates the partitioning of the interface.

3.4 Nonfunctional requirements

In this section a description of the nonfunctional requirements concerning the IRW are provided. A nonfunctional requirement is a user visible constraint on the system that are not directly related with the functional requirement [6].

3.4.1 User interface and human factors

Since the end users of the IRW are the architects who are accustomed to terminal based applications, i.e., applications without a graphical user interface (GUI), there is no actual need for a GUI. However, a GUI is convenient for visualizing the regrouping. Therefore, a simple GUI is preferred.

3.4.2 Output

In contrast with the user interface, the presentation of the output is important. The engineers who perform the actual regrouping of the interfaces will base their actions on the recommendation constructed by the IRW. Therefore, it is necessary that the generated output/recommendation should be easy to read and understand.

3.4.3 Performance characteristics

The IRW should be able to perform reasonably, i.e., output should be generated within a couple of hours at maximum. When a computation requires more than a minute to complete, the IRW should notice the user through a progress indicator. This situation occurs when an interface containing a large amount of symbols is to be regrouped, e.g., interfaces with more than one thousand symbols.

3.5 System models

This section provides the description of the system models. Initially a set of scenarios is given. Using these scenarios, use case models are constructed.

3.5.1 Scenarios

Scenarios are discussed in this subsection. A scenario is a concrete, focused, informal description of a single feature of the system from the viewpoint of a single actor [6]. Three scenarios are identified concerning the IRW. These scenarios are:

- `readInput`
- `regroup`
- `generateOutput`

Table 3.1 presents the scenario for reading the inputs provided by the architect. Table 3.2 presents the scenario for the regrouping process. Finally, Table 3.3 presents the scenario for output generation.

Table 3.1: `readInput` scenario

Scenario name	`readInput`
Participating actor instances	`Jan: Architect`
Flow of events	1. Jan notices that the build time of the TwinScan software system increases. After some analysis Jan is aware that the increase is caused by interfaces that are not IMAS compliant. 2. Jan uses the dependency data and the scope set files to initiate the IRW. 3. The IRW reads the input and parses the dependency relation file and the scope set into its memory/database.

Table 3.2: `decoupling` scenario

Scenario name	`regrouping`
Participating actor instances	`Jan: Architect`
Flow of events	1. Jan starts the IRW. 2. Jan selects an interface to be regrouped. 3. The IRW shows all the symbols defined in the current interface. 4. Jan commands the IRW to begin the regrouping process 5. Jan decides that the result does not represent a correct grouping of symbols and perform manual modifications.

3.5.2 Use case models

From the scenarios described in the previous section use cases can be constructed. Since a scenario is an instance of a use case, we can abstract the scenario to obtain the use cases. A use case is initiated by an actor and it may interact with other actors. A use case represents a complete flow of events in the sense that it describes a series of related interactions that result from the initiation of the use case [6]. The use cases are presented in a tabular form in Tables 3.4, 3.5 and 3.6, and their corresponding UML use case model in Figure 3.2.

Table 3.3: `generateOutput` scenario

Scenario name	`generateOutput`
Participating actor instances	`Jan: Architect`
Flow of events	1. Jan is satisfied with the view of grouping presented by the IRW. 2. Jan commands the IRW to create a recommendation. 3. The IRW generates the recommendation.

Table 3.4: `readInput` use case

Use case name	`readInput`
Participating actor	Initiated by `Architect` Communicates with `CScout`
Entry condition	-
Flow of events	1. `Architect` generates dependency data using `CScout`. 2. `Architect` uses the dependency data and scope set files as input for `IRW`. 3. `IRW` reads the data.
Exit condition	The data is stored in `IRW`'s memory/database.

Table 3.5: `regrouping` use case

Use case name	`decoupling`
Participating actor	Initiated by `Architect`
Entry condition	The data is stored in `IRW`'s memory/database.
Flow of events	1. `Architect` selects an interface to be regrouped. 2. `IRW` shows all symbols defined in current interface. 3. `Architect` starts the regrouping process.
Exit condition	Symbols are regrouped complying to the IMAS criteria.

Table 3.6: `generateOutput` use case

Use case name	`generateOutput`
Entry condition	Symbols of an interface are already regrouped.
Flow of events	1. `Architect` chooses to create an output. 2. `IRW` generates the output.
Exit condition	Grouping information is presented in a output, i.e., textual, charts or graph.

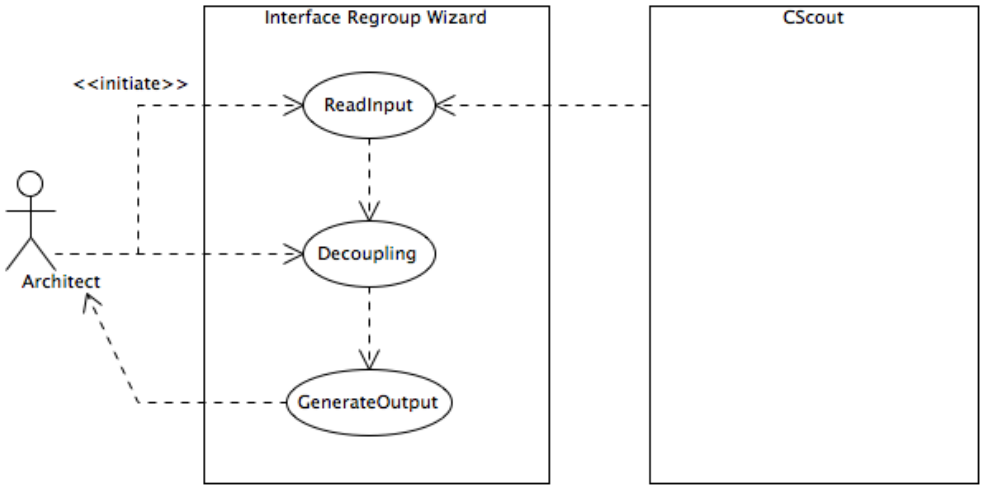

Figure 3.2: Use case model of the IRW.

Chapter 4

Background

In this chapter we discuss some background knowledge that is useful for the development of the IRW. First we briefly discuss a tool, which is being used at ASML for analyzing the source code of the TwinScan software system. This tool is called CScout and the discussion is presented in Section 4.1. We continue with the study of several works which are closely related to the IRW in Section 4.2. Sections 4.3 and 4.4 discuss two techniques for regrouping that are applicable for the IRW. We complete this chapter with a conclusion in Section 4.5.

4.1 CScout

CScout is a source code analyzer and refactoring browser [26] for collections of C programs. The analysis process involves preprocessing, parsing, and semantic analysis of C code. The analysis CScout performs takes into account the identifier scopes introduced by the C preprocessor and the C language proper scopes and namespaces. CScout can store a complete and accurate representation of the code and its identifiers in a relational database. After the source code analysis, CScout is able to, among others, process sophisticated queries on identifiers, files, and functions.

CScout has been successfully applied on large projects containing millions lines of codes, such as the Linux and FreeBSD kernel. At ASML, CScout is used to analyze the source code of the TwinScan software systems. This analysis enables ASML to perform calculations needed to determine the IMAS compliancy of interfaces. The measurements set by the IMAS project, e.g., dependency, dealing and encapsulation, can be calculated using the analysis result of CScout. For example, by performing a query on the analysis result we are able to collect all dependent components and interfaces of a certain interface. ASML has built an interface browser that uses the analysis results to display detailed information on all interfaces. The interface browser also allows us to immediately spot interfaces that are not IMAS compliant.

CScout's ability to precisely discover all users of a certain symbol, serves as the starting point for our research project. The extracted data from the analysis performed by CScout is provided in a textual form where each line declares a user and the symbol it uses. To be more exact, the syntax of a single line in the extracted data is as follows:

```
DEF-DEP user file { [symbol]+ }
```

Here, a `user` can either be the name of a .c file, .ci file or .h file using the symbols between the brackets. A `file` is the name of a .ci file or .h file defining the symbols between the brackets. A `symbol` is the name of the symbol being used by `user`. The plus sign indicates that there may be one or more symbols from `file` that are being used by `user`. When `user` does not use any symbol from `file`, then there exists no such line in the extracted data.

The total size of the extracted data is currently around 100 megabytes and it contains 480.880 lines. Furthermore, the extracted data defines 564.533 symbols and 36.213 users. The exact total number of use dependencies are 1.673.754, i.e., there are 1.673.754 dependencies between user and symbol. So, a symbol is averagedly used by three users. It is our task to interpret and use these enormous amount of use dependency data for the IRW.

4.2 Related Work

Before we actually begin pondering about an approach that is able to tackle the problem stated earlier, we explored other research that is closely related. The works we encountered and discussed below are related to each other in one aspect, i.e., classification. Classification, in the most general terms, is a process of giving names to a collection of objects which are thought to be similar to each other in some respect [9]. Although the focal point of the studies discussed seems to be different, they are related in that they classify entities, such as modules, objects, classes, functions, etc.

Marco Glorie investigated how the software archive for MR-scanners at Philips could be splitted [12]. The software archive of the MR-scanner consists of approximately 9 million lines of code and it is distributed in around 30.000 files. Glorie uses a view-driven architecture reconstruction process called Symphony [33]. Furthermore, Glorie discusses two analysis techniques that could be used to calculate a new architecture [12], i.e., formal concept analysis and cluster analysis.

Tilley et al. present an overview of academic papers that report the application of formal concept analysis to support software engineering activities [31]. They concluded that the majority of work has been in the areas of detailed design and software maintenance.

Snelting and Tip investigated the reengineering of class hierarchies using concept analysis [29]. This work is a prolongation of Snelting's previous work on reeingineering configuration structures from source code [28]. Snelting recognizes that the design of a class hierarchy may be imperfect. For example, different instances of class C may access different subsets of C's members. This is an indication that it might be appropriate to split C into multiple classes. The approach used by Snelting is composed of two steps, i.e., (1) creation of a table that precisely reflects the usage of the classes in a class hierarchy, and (2) creation of a *concept lattice* from the usage table. A concept lattice is a graph illustrating the relationship between objects based on their attributes [36]. The generated concept lattice provides valuable insight into the design of the class hierarchy and it can be used to reach agreement on the design.

Related to Snelting's work, Siff and Reps [27] describe a technique for identifying modules using *concept partitions*. Tonella on the other hand proposes a technique called *concept subpartitions* for module restructuring [32]. We will discuss both techniques and formal concept analysis in detail in Section 4.3.

Next to formal concept analysis, another technique called cluster analysis is able to group a set objects together based on their attributes/characteristics. Andreu and Madnick applied the partitioning concept to a database management system in order to minimize coupling between a set of requirements [5]. First they identified the requirements and the interdependencies between those requirements. The identified requirements and their interdependencies are then converted to a graph problem. The resulting graph is partitioned several times and the alternative partitioning with the lowest value of coupling is the optimal partitioning.

Mancoridis et al. developed and implemented a collection of clustering algorithms for automatic clustering of modules in a software system [24]. This collection of clustering algorithms consists, among others, of the neighbouring partitions algorithm [24] and genetic algorithm [13]. Mancoridis implemented the clustering algorithms in a tool called Bunch [23]. A drawback of Mancoridis' approach is that the computation time for the used algorithms is very high, even for moderate problem size [21]. This drawback has also been recognized by Glorie in his research [12].

Lung presents a study on applying so-called numerical taxonomy clustering to software applications in [21]. Furthermore, Lung proposes an approach to program restructuring at the function level, based on clustering techniques with cohesion as the major concern [20]. The approach provides a heuristic advice in both the development and evolution phase, which help the software designers make decisions on how to restructure a program.

After the examination of the related work, we have come to the conclusion that the problem stated for the IRW is a classification problem. There are two techniques applicable for classification, i.e., formal concept analysis and cluster analysis. There has been a study performed by Van Deursen and Kuipers that tries to compare both techniques in the area of objects identification [34]. They conclude that formal concept analysis outperforms cluster analysis in identifying objects.

In the following sections a detailed presentation of the two techniques, i.e., formal concept analysis and cluster analysis, is provided.

4.3 Formal Concept Analysis

Formal concept analysis has been introduced by Wille [35] and it has been applied in many different fields of research like psychology, sociology, anthropology, biology, medicine, linguistics, mathematics, industrial engineering and computer science. The basic notions of formal concept analysis are those of a *formal context* and *formal concept*. The adjective 'formal' is only used to emphasize that we are dealing with mathematical notions [11]. It is often disregarded in the literature for convenience reasons.

A concept, from a philosophical point of view, is a unit of thoughts consisting of two parts, i.e., the *extension* and the *intension*. The extension covers all objects belonging to this

concept and the intension comprises all attributes valid for those objects [36]. The relation of an object and an attribute, such that it can be said that an object *has* an attribute, is called the *incidence relation*. The objects, the attributes and the incidence relation are combined by Wille in a mathematical definition of a formal context [35].

4.3.1 Formal Context

A formal context $\mathcal{K} := (G, M, I)$, as defined by Wille [11], consists of two sets G and M and a relation I between G and M. The elements of G are the *objects* and the elements of M are the *attributes* of the context. In order to express that an object $g \in G$ is in relation I with an attribute $m \in M$, Wille writes gIm or $(g, m) \in I$ and it is read as 'the object g *has* the attribute m'.

Table 4.1 illustrates a simple example of a context in a tabular form. Here, the animals are the objects and the properties of animals serve as the attributes. The crosses in Table 4.1 indicate relations of animals having properties. An empty cell indicates that an animal does not have the corresponding property. For example, from Table 4.1 we can clearly see that a lion is a preying mammal that does not fly and is not a bird.

Table 4.1: Formal context of animals.

	preying	flying	bird	mammal
Lion	×			×
Finch		×	×	
Eagle	×	×	×	
Hare				×
Ostrich			×	

Wille [11] defines a set of attributes A' common to a set $A \subseteq G$ of objects as:

$$A' := \{m \in M \mid gIm \text{ for all } g \in A\}$$

Correspondingly, Wille defines a set of objects B', which have all attributes in $B \subseteq M$ as:

$$B' := \{g \in G \mid gIm \text{ for all } m \in B\}$$

4.3.2 Formal Concept

A formal concept is a maximum set of objects that all share the same attributes. To be more accurate, a formal concept of the context (G, M, I) is a pair (A, B) with $A \subseteq G$, $B \subseteq M$, $A' = B$ and $B' = A$. A is the extent and B is the intent of the concept (A, B) [11].

To illustrate the notion of a concept, we can use the example given in Table 4.1. For example, when we look at the attributes of the finch, we can ask for all animals having the attributes of the finch. The result is a set consisting of finch and eagle. This result set A of objects is closely connected to the set B of attributes consisting of flying and bird: A is the set of all objects having all the attributes of B and B is the set of all attributes, which are

valid for all objects of A. The extent of a concept determines its intent and the intent of a concept determines its extent.

4.3.3 Subconcept, Superconcept and Concept Lattice

Between the concepts of a given context there is a natural hierarchical order, defined by the *subconcept-superconcept* relation. Suppose (A_1, B_1) and (A_2, B_2) are concepts of a context, (A_1, B_1) is called a *subconcept* of (A_2, B_2), provided that $A_1 \subseteq A_2$, which is equivalent to $B_2 \subseteq B_1$ (proof see [11]). In this case, (A_2, B_2) is a *superconcept* of (A_1, B_1), and it can also be written as $(A_1, B_1) \leq (A_2, B_2)$. The relation $\leq$ is called the *hierarchical order* of the concepts. The set of all concepts of the context (G, M, I) are ordered using the hierarchical order and it is called *concept lattice*.

As an example, in Table 4.1 the preying flying birds describe a subconcept of flying birds. The subconcept in this example consists of eagle as its extent and the attributes preying, flying and bird as its intent. The superconcept consists of the objects eagle and finch as its extent and the attributes flying and bird as its intent. Subsequently, we are able to construct all concepts of the animal context and visualize the hierarchy of all concepts in an undirected graph, which is called as concept lattice. Figure 4.1 illustrates the concept lattice of the animals context in Table 4.1.

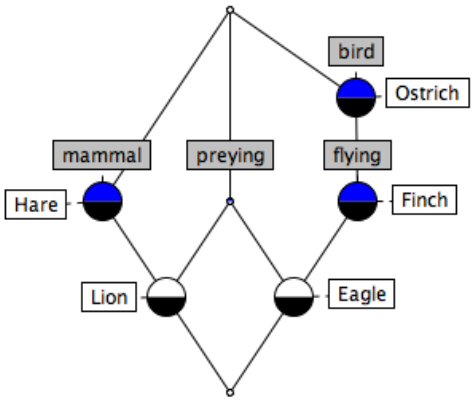

Figure 4.1: Concept lattice of animals.

A concept lattice consists of nodes, lines connecting nodes, and names of all objects and attributes of the given context. The nodes represent the concepts. The lines represent the subconcept-superconcept relation between concepts. Figure 4.1 is generated by a formal concept analysis tool called Concept Explorer [3]. The different sizes of the nodes is due to the fact that, in Concept Explorer, a node is 'fatter' when it contains more objects. This

feature is user-specific and it can be disabled if desired. The blue nodes in Figure 4.1 represent nodes that have attributes. The objects names are presented in the white boxes and the attributes names are presented in grey boxes. The concept lattice can be read in two different ways:

- Top-down, i.e., for an attribute m follow all paths downwards to identify the set G of objects having attribute m.
- Bottom-up, i.e., for an object g follow all paths upwards to identify the set M of attributes of g.

For example, eagle has the attributes preying, flying and bird when we traverse all paths upwards starting from the node containing the eagle object. The other way, for the preying attribute, we collect the objects lion and eagle when we traverse all paths downwards starting from the node containing the preying attribute.

4.3.4 Partitions and Subpartitions

Earlier, we mentioned the notion of *concept partitions* and *concept subpartitions* defined by Siff and Reps, and Tonella, respectively. Siff and Reps uses the notion of concept partition for software modularization purposes. They define a concept partition as a collection of modules that are disjoint, but include all the functions in the input code. Formally, a concept partition of context (G, M, I) is a set of concepts of which their extents are nonempty and form a partition of G. $\mathcal{CP} := \{(A_1, B_1), ..., (A_n, B_n)\}$ is a concept partition *iff* the extents (objects) of the concepts in $\mathcal{CP}$ cover the object set G and are pairwise disjoint [27]:

$$\bigcup_{i=1}^{i=n} A_i = G \text{ and } A_i \cap A_j = \emptyset \text{ for } i \neq j$$

If we apply the the notion of concept partitioning to the animals example, we obtain two concept partitions, which are presented in Table 4.2. Generally, when there is no overlap between a set of concepts, the number of concept partitions will grow exponentially with respect to the number of concepts: The number of concept partitions equals the number of ways to partition a set S of n elements into $k - partitions$ of disjoint, nonempty subsets [24, 12]. This number $S_{n,k}$ is also called the Stirling numbers of the second kind [4]. Suppose we have 5 non-overlapping concepts. This will result in 52 concept partitions.

Table 4.2: Concept partitioning of animals.

$\mathcal{CP}_1$	(*all objects*, $\emptyset$)
$\mathcal{CP}_2$	({Ostrich, Finch, Eagle}, {bird}), ({Hare, Lion}, {mammal})

Tonella stated that concept partitions introduce an overly restrictive constraint on concept extents by requiring that their union covers all objects in the context [32]. As an example, suppose that there exists an object with no corresponding attribute or in other words,

there exists an extent without any intent. The consequence is that concept partitioning returns only a single partitioning, i.e., the partitioning with all objects and no attributes. When concepts are disregarded because they cannot be combined with other concepts, important information identified by formal concept analysis is lost. Therefore, Tonella introduces the notion of concept subpartition where the overly restrictive constraint is removed. A concept subpartition associated with a given context is a set of concepts with disjoint extents. Formally, $\mathcal{CSP} = \{(A_1, B_1)\}, ..., \{(A_n, B_n)\}$ is a concept subpartition *iff* [32]:

$$A_i \cap A_j = \emptyset \text{ for } i \neq j$$

We can also say that concept partitions are a subset of concept subpartitions where the union of the extents in concept partitions are the set of all objects.

For example, using the animals context in Table 4.1 we can retrieve the concept subpartitions presented in Table 4.3. As we can see here, the amount of concept subpartitions explodes when compared to concept partitions.

Table 4.3: Concept subpartitioning of animals.

$\mathcal{CSP}_1$	(*all objects*, $\emptyset$)
$\mathcal{CSP}_2$	({Hare, Lion}, {mammal})
$\mathcal{CSP}_3$	({Lion}, {mammal, preying})
$\mathcal{CSP}_4$	({Ostrich, Finch, Eagle}, {bird})
$\mathcal{CSP}_5$	({Finch, Eagle}, {bird, flying})
$\mathcal{CSP}_6$	({Eagle}, {bird, flying, preying})
$\mathcal{CSP}_7$	({Ostrich, Finch, Eagle}, {bird}), ({Hare, Lion}, {mammal})
$\mathcal{CSP}_8$	({Finch, Eagle}, {bird, flying}), ({Hare, Lion}, {mammal})
$\mathcal{CSP}_9$	({Eagle}, {bird, flying, preying}), ({Hare, Lion}, {mammal})
$\mathcal{CSP}_{10}$	({Ostrich, Finch, Eagle}, {bird}), ({Lion}, {mammal, preying})
$\mathcal{CSP}_{11}$	({Finch, Eagle}, {bird, flying}), ({Lion}, {mammal, preying})
$\mathcal{CSP}_{12}$	({Eagle}, {bird, flying, preying}), ({Lion}, {mammal, preying})
$\mathcal{CSP}_{13}$	({Eagle, Lion}, {preying})

Both techniques of cluster partitions and subpartitions are not suitable for the implementation of the IRW, due to the overly restrictive constraint, and the exponential numbers of resulting subpartitions, respectively.

4.4 Cluster Analysis

Formal concept analysis is a relatively new classification technique. Previously, automatic classification was performed through cluster analysis. Cluster analysis or clustering is the unsupervised classification of patterns, e.g., observations or data items, into groups (clusters) that are similar to each other along one or more dimensions [10]. Cluster analysis has been studied in numerical taxonomy where observations are grouped and segregated based

on numeric measures of similarity or dissimilarity. The measures and strategies for applying cluster analysis are diverse. An overview of several clustering techniques have been provided in [17, 30]. This section provides a brief discussion of existing, most commonly used clustering techniques.

4.4.1 Hierarchical

Hierarchical clustering is a leveled clustering method where at each step two clusters, which are closest or farthest to each other, are merged in agglomerative hierarchical clustering or divided in divisive hierarchical clustering, respectively [18]. When we use the notion of hierarchical clustering in the remaining of this thesis, we actually mean the agglomerative hierarchical clustering. The hierarchical clustering scheme has the following arrangement [19]:

- We have n objects.
- We have n clusterings, $C_1, ..., C_n$. C_1 is the clustering where each object is a cluster, also called *weak clustering*. C_n is the clustering where all objects are in the same cluster, also called *strong clustering*.
- For each clustering C_i there is a corresponding numerical cluster value α_i, which can be defined, for instance, as the closest distance between two clusters in the current clustering.
- We require that the clustering values increase: $\alpha_i \leq \alpha_{i+1}$ for $i = 1, .., n-1$.
- We require that the amount of clusters in the clustering decreases: $\mid C_i \mid \geq \mid C_{i+1} \mid$ for $i = 1, ..., n-1$.

In a hierarchical clustering there are initially n clusters each containing a single object. After merging the closest pair at the first step there are $n-1$ clusters. The merging continues until there is one big cluster left. So, the total number of steps is $n-1$.

An important notion in hierarchical clustering is the *distance* between clusters. The distance between clusters determines which two clusters should be merged. There are several distance measurements discussed in the literature. A couple of examples are [25]:

- *Single-link*: The distance between any two clusters is the minimum distance between each pair of points such that each pair has a point in both clusters.
- *Complete-link*: The distance between any two clusters is the maximum distance between each pair of points such that each pair has a point in both clusters.
- *Average-link*: The distance between any two clusters is the average distance between each pair of points such that each pair has a point in both clusters.
- *Median-link*: The distance between any two clusters is the median distance of all distances between each pair of points such that each pair has a point in both clusters.

Hierarchical clustering is usually visualized by a *dendogram*. Figure 4.2 illustrates such a dendogram. Figure 4.2 shows clearly that B and C are merged first, followed by D and E, etc. The similarity value between two clusters decreases the higher we ascend the hierarchy. The dashed line illustrates an example of a 'good' clustering consisting of three clusters, i.e., (A, B, C), (D, E) and (F, G).

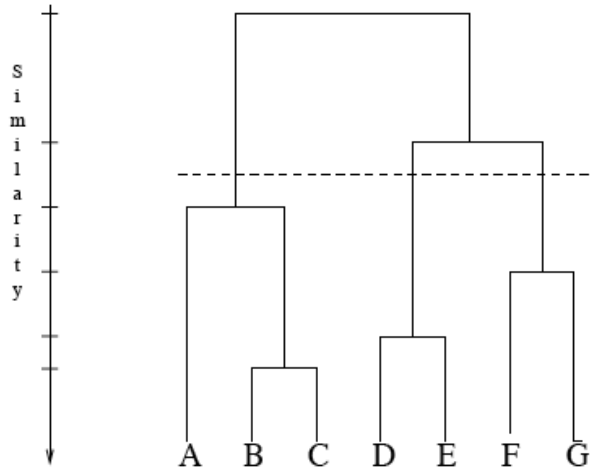

Figure 4.2: Dendogram.

The quality of each clustering calculated by hierarchical clustering can be assessed through cluster validation techniques. We discuss these techniques in further detail in Section 5.1.3.

4.4.2 Partitional

Partitional clustering differs fundamentally from hierarchical clustering. Instead of calculating the entire clustering structure, e.g., a dendogram, a partitional clustering calculates a single partition of the data. As a result, partitional clustering is preferred when dealing with large amount of data. The partitional clustering usually produces clusters by optimizing a criterion function. The partitional clustering algorithm typically runs multiple times with different starting states before determining the best clustering [17].

The k-Means clustering algorithm is the most commonly used algorithm for partitional clustering. The k-Means clustering algorithm groups objects into k partitions. The main idea is to define k centroids, i.e., the center points of k clusters. The next step is to take each object in the dataset and associates it to the nearest centroid. When all objects have been assigned to a centroid, we move the k centroids to the new centroid objects of each cluster calculated in the previous step. We iterate this process until the centroids do not move anymore. Here, the criterion function to be minimized is the squared error function

[22]:

$$V = \sum_{i=1}^{k} \sum_{x_j \in S_i} \mid x_j - \mu_i \mid^2$$

where k is the amount of clusters, S_i is a cluster for $i = 1, 2, .., k$ and μ_i is the centroid of cluster S_i.

4.4.3 Fuzzy

Previously discussed clustering algorithms are categorized as *hard clustering* where each object belongs to one and only one cluster. *Fuzzy clustering* is an extension to hard clustering. In fuzzy clustering each object is associated with every cluster with a membership function [37]. Similar to the partitional clustering, fuzzy clustering produces clusters by optimizing a criterion function and running multiple times until it converges to a stable state [17].

Fuzzy c-means is the most commonly used algorithm for fuzzy clustering. The general description is similar to a k-means clustering algorithm, i.e., it iterates the process of c centroids movements until stability is reached. Additional to the fuzzy c-means is the use of a membership's degree u_{ij} of each object i in a cluster j, where $0 \leq u_{ij} \leq 1$. The higher a membership's degree u_{ij}, the more probable an object i is in cluster j. The criterion function to be minimized for fuzzy c-means is [37]:

$$V = \sum_{i=1}^{N} \sum_{j=1}^{C} u_{ij} \mid x_i - \mu_j \mid^2$$

where N is the amount of objects, C is the amount of clusters, u_{ij} is the degree of membership of x_i in cluster j and μ_j is the centroid or mean point of cluster j. The degree of membership u_{ij} and the centroid μ_j are updated regularly by:

$$u_{ij} = \frac{1}{\sum_{k=1}^{C} \left(\frac{|x_i - \mu_j|}{|x_i - \mu_k|} \right)^{\frac{2}{m-1}}} \text{ and } \mu_j = \frac{\sum_{i=1}^{N} u_{ij} \cdot x_i}{\sum_{i=1}^{N} u_{ij}}$$

The iteration process has reach a stable state when u_{ij} does not change significantly.

4.4.4 Evolutionary

Evolutionary clustering is motivated by natural selection. It uses evolutionary operators and a population of solutions to obtain the globally optimal partition of the objects. The most commonly used operators are selection, recombination and mutation. Each transforms candidate solutions into other solutions. A fitness function determines the likelihood of survival of the solutions. Generally, an evolutionary clustering algorithm can be described as follows [17]:

1. Choose a random population of solutions and associate a fitness value for every solution, e.g., using squared error function as the fitness function, a small squared error value will have a larger fitness value.

2. Use evolutionary operators to generate next population of solutions. Reevaluate the fitness value for each solution. Repeat this step until a threshold is reached.

The best known examples of evolutionary clustering algorithms are genetic algorithms. Doval proposed a clustering approach that uses a genetic algorithm [8]. The algorithm starts with a random clustering and tries to compute the optimal clustering by applying selection, mutation and the popular crossover operators. A detailed presentation on genetic algorithm is presented in [8, 24].

4.5 Conclusion

In this chapter we have reviewed a tool called CScout that is able to collect the required data for the IRW. Furthermore, we have discussed several works that are closely related. After this discussion, we concluded that the problem defined for the IRW is a classification problem. We also reviewed two techniques that can be used for tackling the classification problem, i.e. formal concept analysis and cluster analysis.

From the research performed by van Deursen et al. [34] we obtained information that formal concept analysis is superior when compared to cluster analysis. After a review of formal concept analysis, we conclude that formal concept analysis is not suitable for regrouping the interface symbols. A concept lattice for several hundreds of objects and attributes, which is common in our case, is not readable and comprehensible. Using concept partitions we obtain partitions that might disregard important information. Using concept subpartitions we obtain a huge set of subpartitions. Glorie confirmed our conclusion in his work on the splitting of the software archive for MR-scanner at Philips Medical [12].

Therefore, we looked at the alternative, i.e., cluster analysis. From the review we have performed, we concluded that partitional and fuzzy clustering are not suited due to the requirement of an initial amount of k clusters. We do not have the information about in how many clusters the symbols should be regrouped and which amount is the optimum. To retrieve this information, we have to execute the mentioned clustering algorithms on all possible k values, i.e., from 1 to n, where n is the number of symbols. From all the possible clusterings, we can calculate the optimal value of k. This process is expensive and therefore not suited. Furthermore, Glorie has investigated the possibilities of genetic clustering algorithms in splitting the archives. He concluded that genetic clustering algorithms are hindered by scalability issues [12]. Scalability is an issue also for the IRW.

One option is left: hierarchical clustering. We will explore this option in the continuation of this thesis.

Chapter 5

Interface Regroup Wizard

After a study of related works and applicable techniques discussed in the previous chapter, we implement a prototype application called the IRW that is able to regroup the interface symbols such that the generated groups are IMAS compliant. This chapter provides a description of the used algorithm, the encountered challenges and the development process.

5.1 Algorithm

The algorithm we use for the IRW is the agglomerative hierarchical clustering algorithm as presented by Johnson [19] and discussed earlier in Section 4.4.1. The objects we have to cluster are the symbols in an interface. In order to cluster the symbols, we have to determine some kind of distance measure.

5.1.1 Distance

A popular distance measure we will use for the IRW is the *Minkowski metric* [17]. It defines the distance δ_p between elements s_i and s_j as:

$$\delta_p(s_i, s_j) = \left(\sum_{k=1}^{d} \mid s_{i,k} - s_{j,k} \mid^p \right)^{\frac{1}{p}}$$

where d is the dimensionality of the data. The *Manhattan* and *Euclidean* distances are special variants of the Minkowski metric where $p = 1$ and $p = 2$, respectively. The Euclidean distance is the most popular of the two.

In our case, the only information we have about the symbols are its users. This indicates that the dimensionality of the Minkowski metric is equal to one. Therefore, it does not matter which value for p we choose. However, this is quite sufficient to calculate the distance between two symbols. We have determined that the distance between two symbols is the number of users they have in common relative to the total number of users of both symbols. Using the Minkowski metric we infer the following distance metric between two symbols:

$$\delta(s_i, s_j) = 1 - \frac{\mid U_i \cap U_j \mid}{\mid U_i \cup U_j \mid} \tag{5.1}$$

where s_i and s_j are symbols and U_i and U_j are sets of users that uses symbols s_i and s_j, respectively. A value of 0 indicates two symbols having identical users sets and a value of 1 indicates two symbols having distinct users sets. For example, suppose we have two symbols s_1 and s_2 and their corresponding sets U_1 and U_2 of users:

- $U_1 : \{a, b, c, d\}$
- $U_2 : \{b, c, e, f\}$

Using equation 5.1 the distance between s_1 and s_2 is $\frac{2}{3}$.

Equation 5.1 enables us to calculate the distance between symbols. A hierarchical clustering algorithm also requires a distance measure for measuring the distance between two clusters. In Section 4.4.1 we already discuss four methods that can be used to measure the distance between two clusters, i.e., single-link, complete-link, average-link and median-link. Formally, the distance between two clusters c_i and c_j can be calculated as follows:

- $\delta_{single}(c_i, c_j) = \min_{1 \leq n \leq N, 1 \leq m \leq M} \delta(s_n, s_m)$
- $\delta_{complete}(c_i, c_j) = \max_{1 \leq n \leq N, 1 \leq m \leq M} \delta(s_n, s_m)$
- $\delta_{average}(c_i, c_j) = \frac{1}{N \cdot M} \sum_{n=1}^{N} \sum_{m=1}^{M} \delta(s_n, s_m)$
- $\delta_{median}(c_i, c_j) = \text{med}_{1 \leq n \leq N, 1 \leq m \leq M} \delta(s_n, s_m)$

where N and M are the numbers of symbols in respectively c_i and c_j. The function med is a median function that returns the middle distance value for an odd numbers of distance values, or the average of the two middle values for an even numbers of distance values. The behaviour, i.e., the resulting clusterings, of each distance measure specified above differs. A hierarchical clustering algorithm that uses a single-link method produces a few large clusters and many small clusters. The complete-link method on the other hand produces compact clusters of comparable size. A major drawback of the single-link method is that it might group two symbols together that are *not* similar, because it selects the closest distance value and disregards other distance values [17]. The average-link and median-link methods are basically compromised methods of single-link and complete-link methods, i.e., the combination of the best of the two methods.

The median-link method is preferred above the average-link method because it prevents the inclusion of outliers in the distance calculation. A drawback of the median-link is that it terminates the clustering process in an earlier phase when compared with the average-link method, i.e., two clusters are recognized as totally distinct in an earlier phase. If we continue clustering after this phase, the hierarchical clustering results in a clustering that is not meaningful, i.e, clusters are randomly selected and merged. Therefore, next to the four methods described earlier, we have devised a method called adaptive-link, which is a combination of the median-link and average-link:

$$\delta_{adaptive}(c_i, c_j) = \left\{ \begin{array}{ll} \delta_{median}(c_i, c_j) & \text{if } 0 \leq \delta_{median}(c_i, c_j) < 1 \\ \delta_{average}(c_i, c_j) & \text{if } \delta_{median}(c_i, c_j) = 1 \end{array} \right\} \tag{5.2}$$

The adaptive-link method continues the clustering process in a meaningful manner by switching to the average-link method. We will implement the five methods described above for comparison reasons.

5.1.2 Hierarchical

Initially, when we wish to cluster a collection of n symbols, we put each symbol in a cluster, generating n clusters (weak clustering). From these n clusters we select two candidate clusters that are closest to each other, with respect to the distance measure used. These two candidates are merged, decreasing the number of clusters to $n-1$. We continue this process until there is a single cluster left containing all symbols (strong clustering).

The pseudocode for the discussed scheme is presented in Algorithm 1. Here, a cluster consists of two elements, i.e., a set of symbols and a distance value of the two last merged clusters. Initially, the distance value is set to zero. At the end of each iteration, after the merged clusters are added, specific actions can be performed such as repainting the dendogram and/or calculating cluster validation values. The time complexity of the hierarchical clustering algorithm is $O(n^2)$ for each merging step at the cluster level, i.e., for each cluster we have to compare it with all other clusters in order to determine the closest neighbouring clusters.

Algorithm 1: Hierarchical clustering algorithm

```
Data: S: Set of symbols
begin
    C: Set of clusters
    foreach s ∈ S do
        c: Cluster
        c.symbols ← {s}
        c.distance ← 0
        C ← C ∪ c
    repeat
        p: Pair of clusters
        d ← max ℝ
        foreach c_i ∈ C do
            foreach c_j ∈ C do
                if i ≠ j & δ(c_i, c_j) < d then
                    d ← δ(c_i, c_j)
                    p ← {c_i, c_j}
        c: Cluster
        foreach c_p ∈ p do
            c.symbols ← c.symbols ∪ c_p.symbols
        c.distance ← d
        C ← (C \ p) ∪ c
        /* do something */
    until | C |= 1
end
```

5.1.3 Cluster Validation

In order to determine in how many groups we have to regroup the interface, we should validate the clustering computed by the hierarchical clustering algorithm. In the literature there are several measurements available that are able to validate a clustering. We have chosen for the combination of the *variance* and *connectivity* [14]. Variance is used to calculate the intra-cluster variance of the distances between the symbols and the centroid of the cluster, i.e, the symbol in the center of the cluster. Using variance we measure the quality of the compactness of the clustering, i.e., a low variance indicates compact clustering. Variance is calculated as

$$Var(C) = \sqrt{\frac{1}{N} \sum_{C_k \in C} \sum_{s \in C_k} \delta(s, \mu_k)}$$

where N is the total number of symbols, C is the set of all clusters, μ_k is the centroid of cluster C_k. The interval of variance is $[0, +\infty]$ and it should be minimized.

Connectivity is used to measure the connectedness of the symbols in the clusters by evaluating the degree to which neighbouring symbols have been placed in the same cluster [14]. The idea is to penalize the situation where neighbours are not placed in the same cluster. The closer the neighbour, the higher will be the penalty. A low connectivity indicates strongly connected clusters. Connectivity is calculated as

$$Conn(C) = \sum_{s \in S} \sum_{j=1}^{L} x_{s, nn_{s(j)}}$$

where

$$x_{s, nn_{s(j)}} = \left\{ \begin{array}{cc} \frac{1}{j} & \text{if } s \in C_k \wedge nn_{s(j)} \notin C_k \\ 0 & \text{otherwise} \end{array} \right\}$$

Here, S is the set of symbols in this clustering. $nn_{s(j)}$ is the jth closest neighbour of symbol s. L is a parameter value determining the number of neighbours participating to the connectivity measure. $x_{s, nn_{s(j)}}$ is actually a penalty value for symbol s and its jth closest neighbour. As we can see, a higher penalty value is given to a more nearby neighbour not in the same cluster. The interval of the connectivity is $[0, +\infty]$ and it should be minimized.

So, a good clustering can be recognized by a low value of variance and connectivity. In a two dimensional chart, a good clustering can be detected by a 'knee' in the chart.

5.2 Challenges

Before we actually start with the design and implementation of the IRW, we have to analyze and overcome existing challenges. In this section we provide an overview of the main challenges we encountered. Furthermore, we also provide possible solutions for every challenge.

5.2.1 Environment

The first challenge is the operating system where the IRW should be able to execute. At ASML, the most used operating systems are Windows and Solaris and the main operating system we work at is OS X. Therefore, we would like to use an operating system independent development platform.

Java provides us with such platform. We will use Java as the programming language for the implementation and Eclipse IDE as the development environment [15].

5.2.2 Scalability

The next and main challenge we have to overcome is scalability. The size of the TwinScan software system is immense. The extracted symbol-user dependency information provided by CScout is also enormous. As mentioned earlier, the size of this extracted dependency data is around 100 megabytes and contains 564.533 symbols. We have attempted to parse this extracted data into memory. This parsing process took approximately one hour and costs more or less half gigabyte of memory. Therefore, it is desired to persist the extracted data such that the initialization time is reduced. By persisting the data we are able to fetch only the symbols we need, instead of loading the entire set of symbols. This method also reduces the amount of memory used.

After considering several alternatives of object relational mapper (ORM)[1], we have decided to use the Hibernate framework to persist the data and MySQL as the underlying database. Hibernate is a popular open source ORM framework in the Java world. Hibernate eases the integration of a relational database into a Java application. MySQL is a popular open source relational database that is also used at ASML. For a detailed description of Hibernate and MySQL, we would like to refer the reader to the documentations provided at their websites[2].

Next to the loading process, the scalability also affects the hierarchical clustering algorithm itself. The $O(n^2)$ complexity of the algorithm at the cluster level in combination with recalculation process of the distances between the symbols at each iteration step will probably cause performance problems. To control this potential problem, we utilize an optimization technique called memoization[3]. A hash-table is used to store the distances between the symbols in order to prevent the distance recalculation process. Instead of a recalculation, a look-up operation is performed. As a result, we expect that the recalculation process does not have a significant impact on the performance anymore. Consequently, the complexity of the algorithm will be the source of any performance problem.

[1] ORM is a technique that converts data types stored in database systems into objects in Object-oriented systems such as Java, and vice versa.

[2] Hibernate: `http://www.hibernate.org/5.html`
MySQL: `http://dev.mysql.com/doc/`

[3] Save (memoize) a computed answer for possible later reuse, rather than recomputing the answer. `http://www.nist.gov/dads/HTML/memoize.html`

5.2.3 Encapsulation

Encapsulation is an important notion at ASML. For example, it is not preferred to cluster a group of symbols that is used at the functional cluster level with a group of symbols that is used at the component level, with the exception of functional coherent symbols. When we allow such grouping, the latter group that is used at component level, becomes visible at functional cluster level. This is not desired. However, when these two groups do share a large group of users, say 90%, we might allow the grouping due to tight coupling. In the end, permission of the responsible architect is required for multilevel grouping. The threshold value that allows multilevel grouping has to be set by the architects also, e.g., suppose a threshold value of 50% is chosen, the architect will be prompted for permission when the IRW attempts to group two clusters with different levels having more than 50% common users. Otherwise, when the common users is below 50%, the multilevel grouping is prohibited. To achieve this we have to slightly modify Algorithm 1.

5.3 Design

After analyzing the algorithm and the main challenges, we proceed with the design of the IRW. Initially, we model the static structure and we continue with the dynamic structure. The modeling language we use is the Unified Modeling Language (UML) 2.0. Currently, UML is the de facto standard in modeling software systems.

5.3.1 Structural Models

To model the structure of the IRW we make use of UML class diagrams. Classes are basically abstractions that specify the common structure and behaviour of a set of objects. UML class diagrams describe the system in terms of objects, classes, attributes, operations and their associations [6].

Persistent

At first, we model the objects in the IRW that are persistable, i.e., we would like these objects to be persisted. These persistable objects are collected in a package we called *persistent*. Figure 5.1 illustrates a class diagram of the persistent package. We have omitted the operations in order to increase readability. The class diagram shown in Figure 5.1 is constructed based on the scope set files provided by ASML. These scope set files describe the hierarchical structure of the TwinScan software as is illustrated in Figure 2.3.

We have further classified the classes in the persistent package into two subpackages called *clusterable* and *cluster*. Basically, the clusterable package contains classes of which their instances are clusterable, i.e., class instances (objects) of which their symbols can be regrouped. Initially, it was our assignment to regroup the symbols of an interface, but we perceived that the architects wish to regroup all symbols in a component instead of a single interface. Therefore, we have decided to classify all objects that contain symbols either directly or transitively, with the exception of users, as clusterable. This construction allows the regrouping of symbols in a building block or higher level when desired. Furthermore,

we utilize the composite design pattern for modeling the users and the symbols those users define and/or use.

The cluster package contains classes of which their instances represent the results of the clustering algorithm. We decided to make these objects persistable because we want to be able to retrieve the result instead of reexecuting the clustering algorithm. Furthermore, the cluster validation measurements discussed in Section 5.1.3 are calculated as soon as a new cluster is created by the hierarchical algorithm. The value of this measurements are stored in the `variance` and `connectivity` variable of the `Clustering` class. When we omit the cluster package, the structure of the persistent package is more or less similar to the hierarchy of the TwinScan software architecture as presented in Figure 2.3.

All class instances (objects) of the classes in Figure 5.1 can be persisted by Hibernate. For the persistence of these objects we require a database schema that corresponds with the structure of the class diagram. Hibernate is able to generate the database schema by means of a set of mapping rules. The set of mapping rules for the persistable classes are generated manually.

The entity relationship diagram shown in Figure 5.2 presents a database schema that is equivalent to the class diagram illustrated in Figure 5.1. All classes in Figure 5.1 are represented by database tables in Figure 5.2 with the exception of `CFile` and `HFile`. For the inheritance persistence of a `User` and its children, `CFile` and `HFile`, we have chosen for a different approach. In this case, `CFile` and `HFile` instances are persisted in the `USER` table. These instances are distinguished by a `USER_TYPE`. For the other cases of inheritance, we create a separate table for each class, e.g., `Clusterable` and its children, `InterfaceProvider`, `Component` and `Interface`.

Note that `n-to-n` relationships are not possible in a relational database. Instead, these relationships are represented by an intermediate table. An example is the `SYMBOL_USER` table. This intermediate table represents an `n-to-n` *use* relationship between users and symbols.

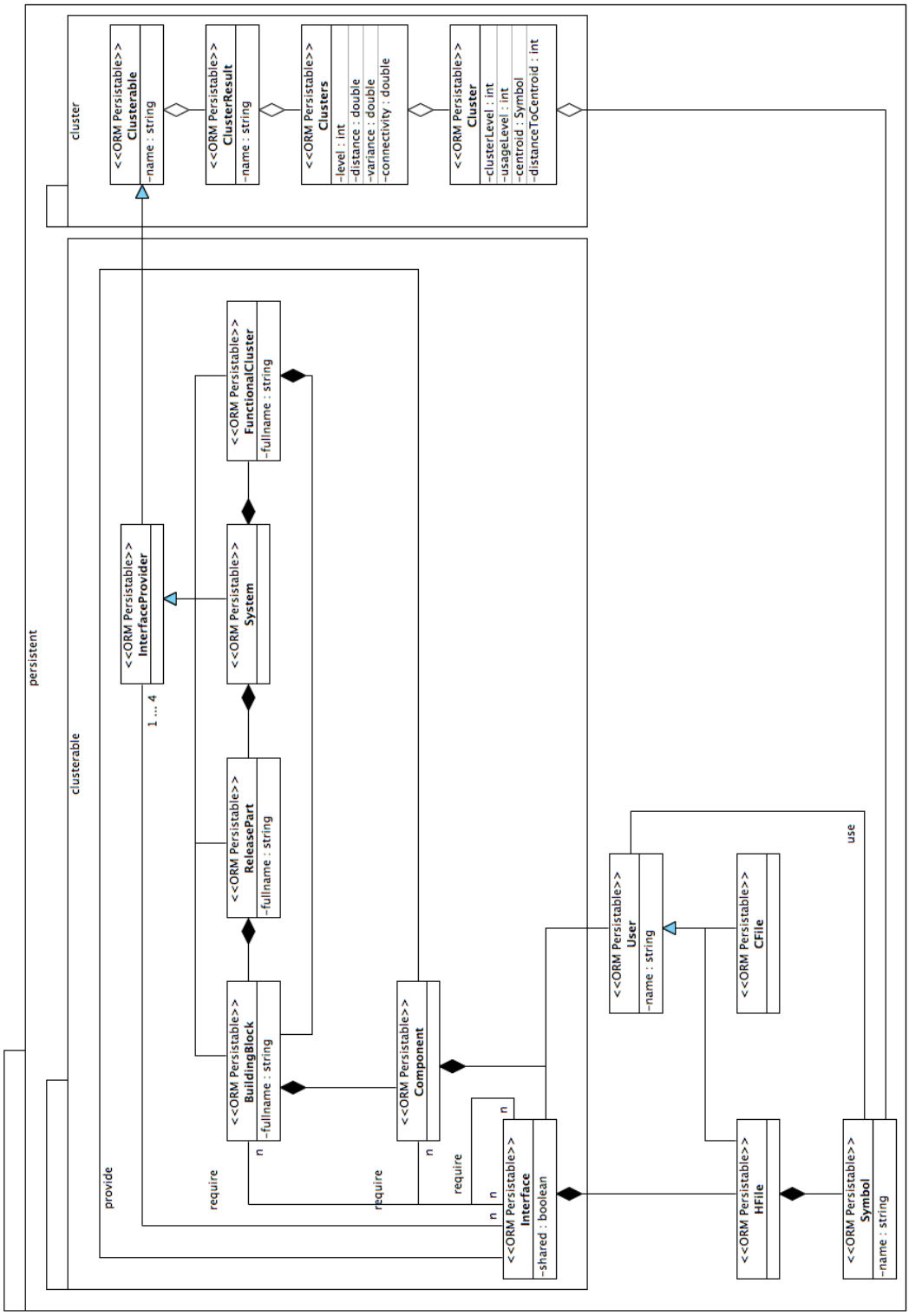

Figure 5.1: UML class diagram for persistent classes.

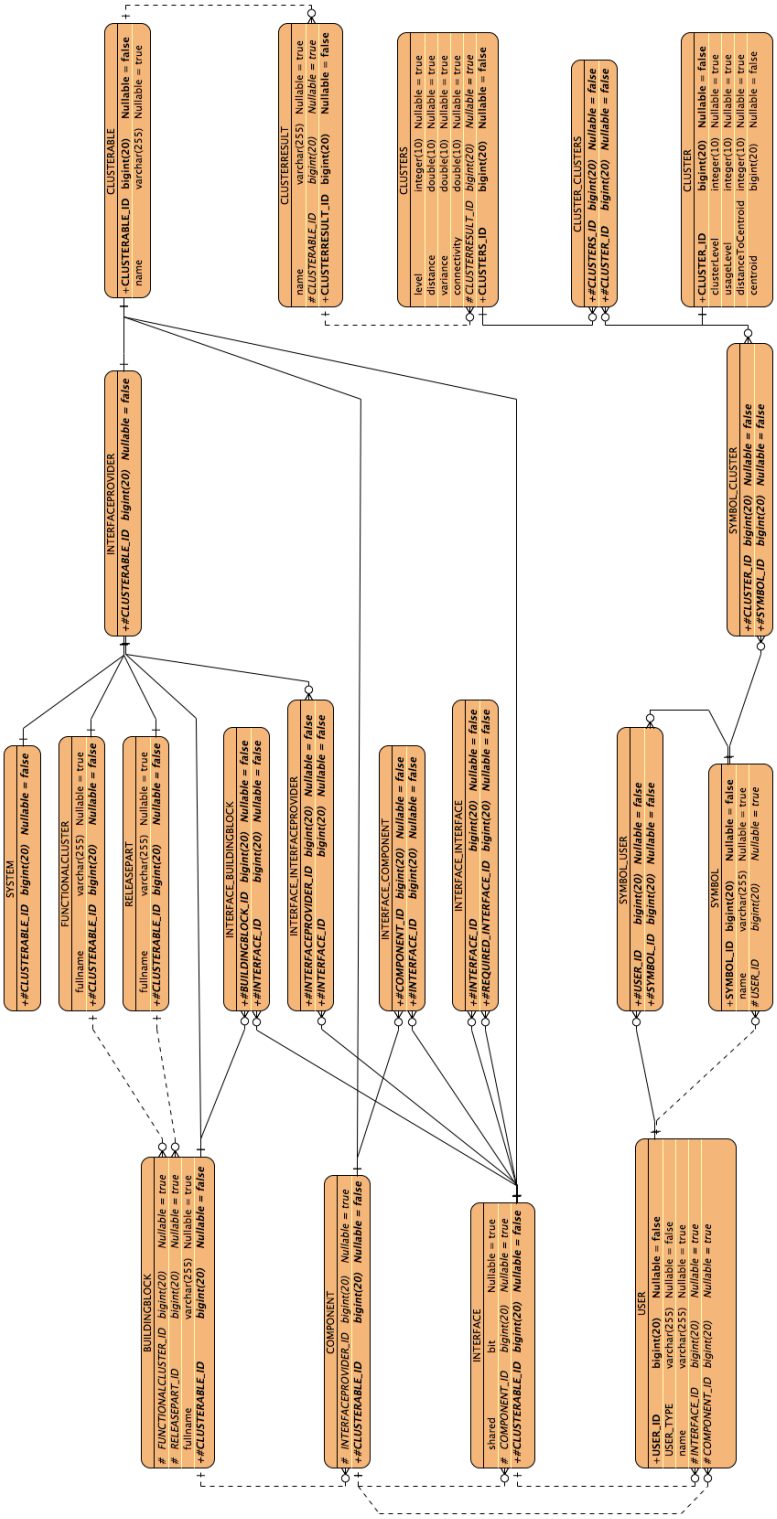

Figure 5.2: Entity relationship diagram of persistable objects.

Hierarchical

For the design of the hierarchical algorithm, we create a separate package called *hierarchical*. Figure 5.3 illustrates the structure of the hierarchical package. Here, we utilize the strategy design pattern. The abstract `Hierarchical` class is the main class that implements the scheme as described in Algorithm 1, i.e., finding and merging the two closest clusters. The five distance measurements discussed in Section 5.1.1 are implemented by its children, i.e., `HCAdaptive`, `HCAverage`, `HCComplete`, `HCMedian` and `HCSingle`.

By isolating the classes that execute the hierarchical algorithm, we obtain a modular construction. Using this construction we are able to easily extend the IRW with other clustering algorithms or techniques, such as concept subpartitions in formal concept analysis.

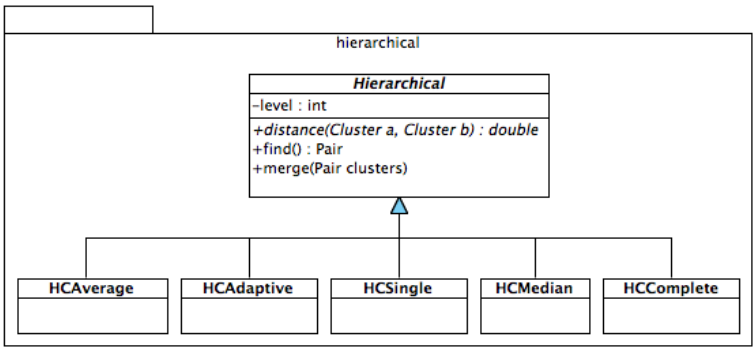

Figure 5.3: UML class diagram for hierarchical classes.

Parser

The *parser* package, which is illustrated in Figure 5.4, contains three classes: `ParserMain` `UseDef` and `ScopeSet`. The `ParserMain` is the main class of `parser` package, which creates both `UseDef` and `ScopeSet` instances.

An instance of the `UseDef` class is able to parse the extracted dependency data obtained from CScout. This dependency data is first read and transformed into objects. Subsequently, all objects are persisted using Hibernate.

An instance of the `ScopeSetParser` class is able to parse the scope set files that define software structure as illustrated in Figure 2.3. Similar to the actions performed by the `UseDefParser` instance, the scope set files are first read and transformed into objects. Subsequently, all objects are persisted using Hibernate.

GUI

For the graphical user interface (GUI), we created several classes. The structure of the GUI is presented in Figure 5.5. The main class for the GUI is the `IRWMain`. This is also the class that will be called when launching the IRW application, i.e., `IRWMain` contains the

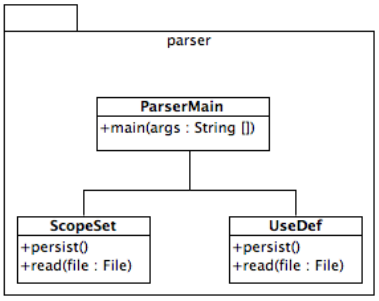

Figure 5.4: UML class diagram for parser classes.

main method. The `IRWMain` class communicates with other dialogs, e.g., `UserDialog` and `ClustersDialog`. These dialogs provide an end-user with information concerning the users using a certain symbol, and the clustering at a certain phase, respectively.

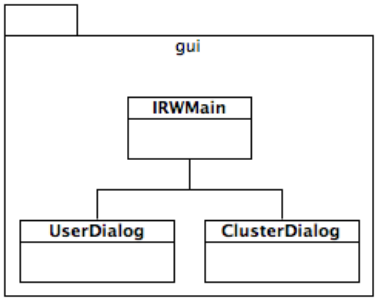

Figure 5.5: UML class diagram for GUI classes.

Utilities

The `Util` package contains utility classes that are required by the IRW. The structure of the `Util` package is illustrated in Figure 5.6. The most important class in the utilities class is the `Globals` class. The `Globals` class contains all global declarations that are visible for all classes in the IRW. These global declarations are, for example, global constants, hash tables and global methods. We plan to use two hash tables. One for storing the distances between symbols, and another for storing a sorted list of neighbouring symbols of all symbols, i.e., hash table of lists. The `neighboursTable` improves the calculation performance of the connectivity values.

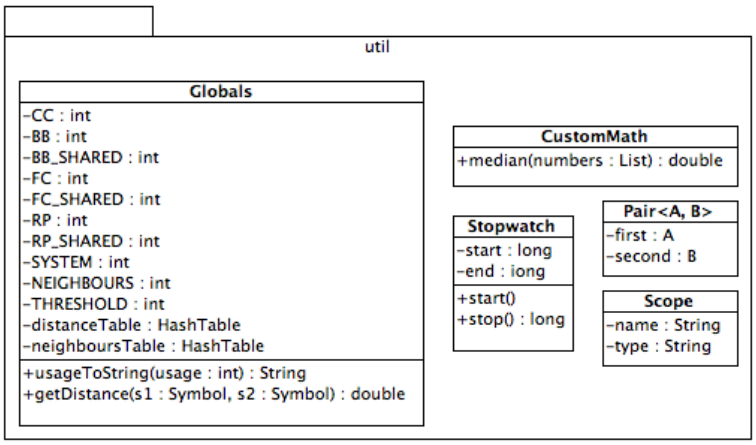

Figure 5.6: UML class diagram for utility classes.

Overview

For the total overview of all packages in the IRW and their associations with each other, we present a package diagram in Figure 5.7. As we can see, the `Hierarchical` package is the heart of the IRW where it is interacting with almost all other packages. The `GUI` package is the external interface of the IRW, allowing end-users to communicate with the IRW.

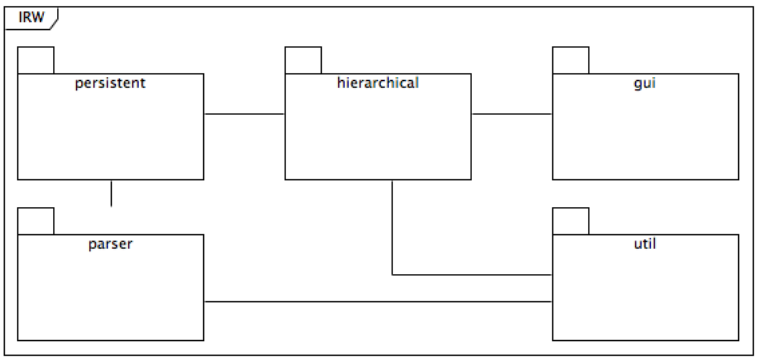

Figure 5.7: UML package diagram of IRW.

5.3.2 Behavioral Models

The structure of the IRW is now captured in several class diagrams and a package diagram. It is now time to model the behaviour and collaboration of these classes.

Finite State Machine

To model the behaviour of the IRW as a whole, we use a UML statechart diagram. A UML statechart diagram is a representation of a finite state machine. A finite state machine is used to model the behaviour of a system and it is composed of states, transitions between the states, and actions corresponding to transitions.

Figure 5.8 shows a UML statechart diagram for the IRW. Initially, the IRW is `started` by the start action. Subsequently, we can load all symbols of an interface. This loading action changes the state of the IRW into `loaded`. In this state, we can either choose for loading other symbols or beginning with the regrouping process. The former does not change the state of the IRW. The latter changes the state into `regrouped`. When we are finished analyzing the result of the regrouping we can return to `loaded` state. From each defined state, we can stop the execution of the IRW by performing the exit action.

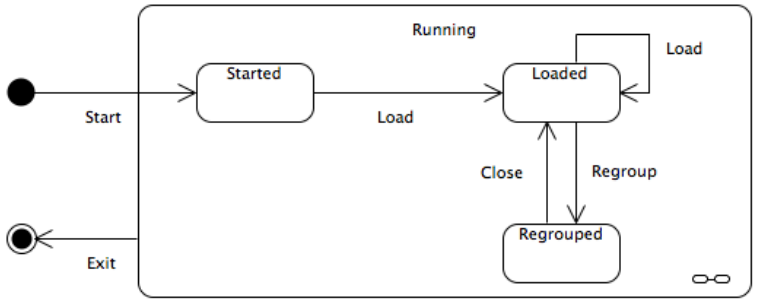

Figure 5.8: UML statechart diagram of IRW.

From the statechart diagram in Figure 5.8 we draw the conclusion that there are three main actions involved, i.e., start, load and regroup. Next to those three actions, we have another action that is crucial for the correct functionality of the IRW, i.e., persistence. The persistence action is not modeled in the statechart diagram, because it is a separate action performed only once, and therefore, not required at each IRW execution.

For each of these actions, we are able to model the interactions of the class instances in the IRW. The interactions or communications between class instances are usually modeled by a UML sequence diagram. In the following subsections we describe each of the actions mentioned.

Persistence

The persistence action is an important process, which has to be performed only once. The persistence process is the parsing and storing process of the data provided by the scope set files and the dependency data file. We have to store the data bottom up to ease the persistence process, i.e., we have to store the symbols first, then the files, interfaces, etc. Therefore, we have to parse and store the dependency data first before continuing with the scope set files. The sequence diagram shown in Figure 5.9 illustrates the described persistence action.

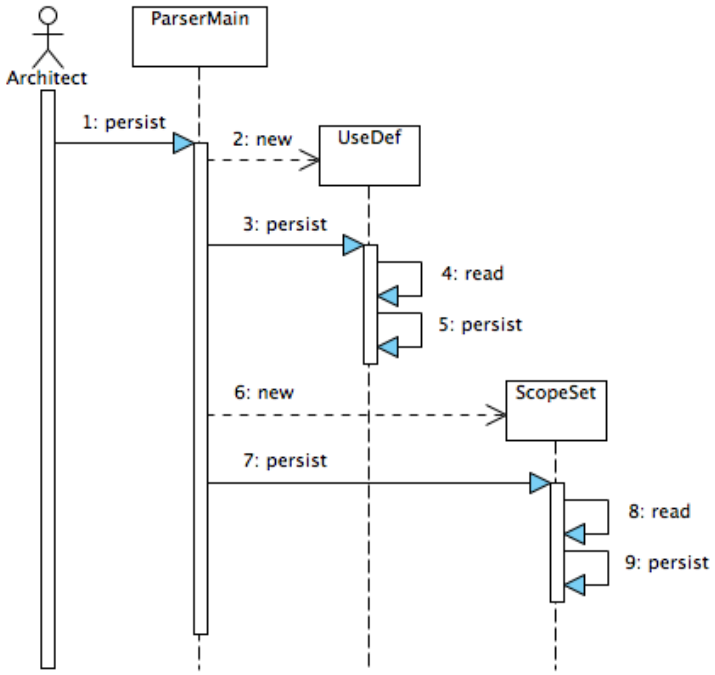

Figure 5.9: UML sequence diagram of persistence.

Start

The start action is basically an action that is required in order to start the IRW. For example, a user can start the IRW by double-clicking the application. The main class `IRWMain` is then called and it initializes the GUI. Figure 5.10 shows a UML sequence diagram illustrating the start action.

Load

The next action we describe is the loading action. By loading we mean the loading process of the symbols of selected interface(s). As described by the first functional requirement in Section 3.3.1, we have to collect all symbols of interface(s), which we are going to regroup. The persisted symbols are fetched from the database using Hibernate. After the loading process is finished, we display the symbols to the user. Figure 5.11 illustrates the loading process.

Regroup

The final and main action of the IRW is the regroup action. When a user chooses to regroup a loaded set of symbols, a new `Hierarchical` instance is created that performs the clustering as described in Algorithm 1. The `Hierarchical` instance requests the `Global`

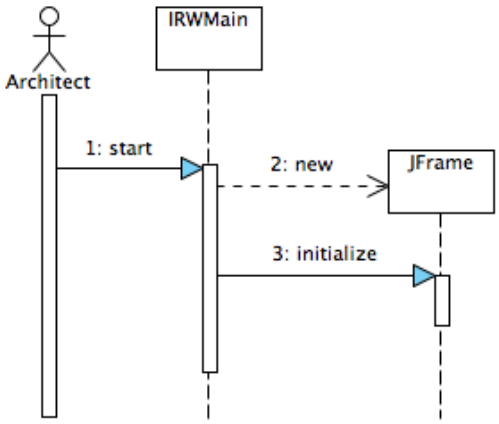

Figure 5.10: UML sequence diagram of start.

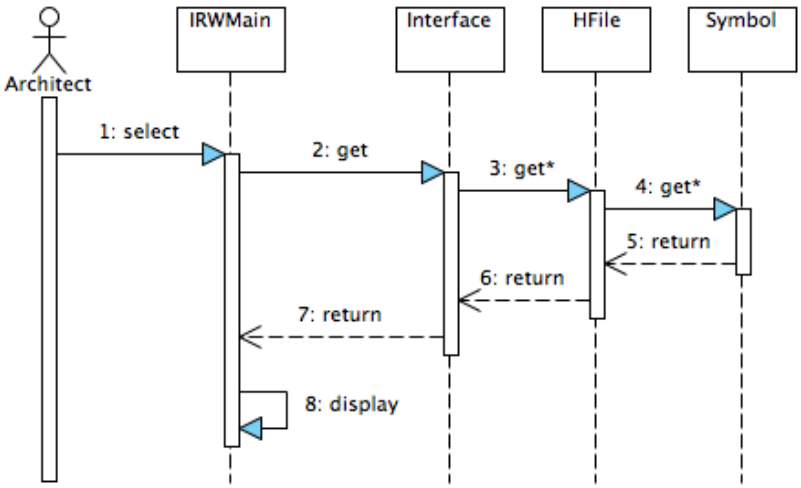

Figure 5.11: UML sequence diagram of load.

class for the distances between the symbols. Here, we use an optimization technique called memoization, i.e., when the distance between two symbols is not present in the hash table in the `Global` class, it is calculated first and inserted into the hash table, before the distance value is returned. The `Hierarchical` instance doesn't see this process. Generally, Figure 5.12 illustrates the behaviour of the regroup action.

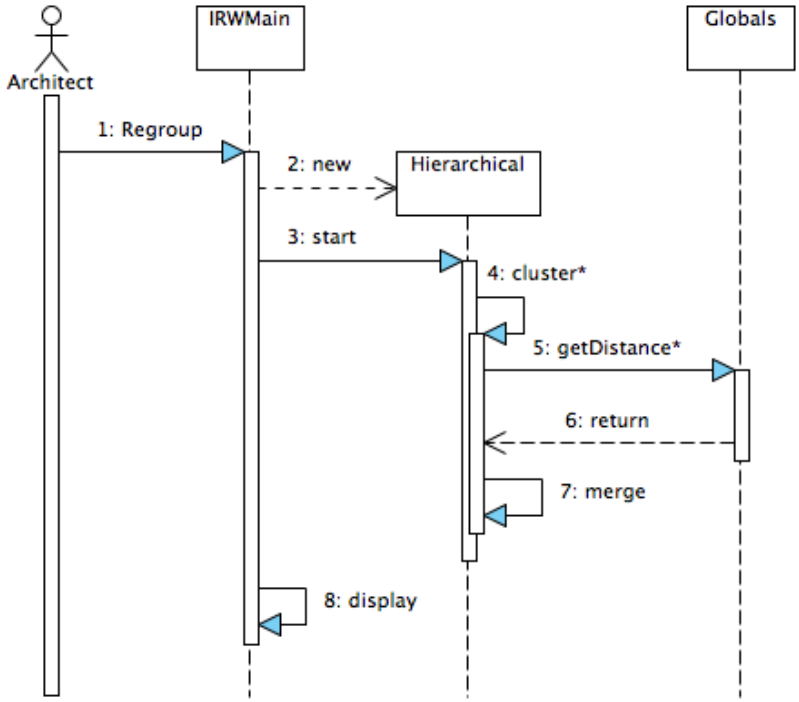

Figure 5.12: UML sequence diagram of regroup.

5.4 Output

For the output of the IRW, the most rational choice would be the generation of a dendogram, because a dendogram is the typical representation used for hierarchical clustering. However, a dendogram illustrating a clustering of several hundreds of symbols would be rather large and incomprehensible. For this reason only, we do not present a dendogram.

Another option is to output a *selected* clustering in its textual form. A drawback of this option is that we do not know which clustering we have to select, i.e., which clustering is optimal. For this problem we have implemented the cluster validation algorithms as discussed in Section 5.1.3. A chart, visualizing the cluster validation values, helps us to decide which clustering is optimal. So, with the assistance of cluster validation, we are able to select the optimal clustering that we can output.

The generated textual output has another drawback, i.e., it does not visualize the relationships between the created clusters. An interesting option that is able to visualize these relationships is a concept lattice. Although, not usable for the actual clustering of the symbols, formal concept analysis is able to efficiently visualize the relationships between objects through a concept lattice. The generated concept lattice, which corresponds to a selected clustering, visualizes clusters as sets of symbols (attributes) that will be used by users (objects). For the results, we refer the reader to Chapter 6.

Chapter 6

Results

This chapter presents the results of the IRW. Initially, we discuss the IRW itself in Section 6.1, i.e., all visible windows and the possible functionalities. Subsequently, we discuss the problems we have encountered in Section 6.2. Finally, we present the results of a case study. For the case study, we have selected one interface.

6.1 Graphical User Interface

The IRW consists of multiple windows, where each window has a different purpose. Figure 6.1 illustrates the main window of the IRW that is shown when the IRW is started. In the main window we can search for an interface or component. We can browse a functional cluster and view its contents, i.e., components and interfaces. The interfaces list shows all interfaces of a selected component and the symbols list shows all symbols of the selected interface(s). Figure 6.2 illustrates the main window that is loaded with symbols. Here, the interface `WLXA` was searched. Subsequently, all symbols of `WLXA` are loaded. As we can see, `WLXA` is located in functional cluster `FC-001: Wafer_Handling` and component `WL`.

In the main window we can choose to regroup the symbols of the selected interface(s) by clicking the Regroup button, to view the users of the selected symbol(s) by clicking the View Users button, and to close the IRW by clicking the Close button. Furthermore, several options can be selected in the Method and Preferences menu bar. In the Method menu bar, we can select which distance measure we plan to use, e.g., single-link, median-link or adaptive-link. In the Preferences menu bar we can select to include struct's elements and/or local users. In Section 6.2 we discuss the reasons for the addition of these preferences.

When the View Users button is clicked the set of users using the selected symbols are presented in a user dialog window. Figure 6.3 illustrates an example of such user dialog window showing the users of symbol `WLXA_swap`.

When the Regroup button is clicked, a new instance of `Hierarchical` is created that executes the hierarchical clustering algorithm using the selected distance measure. During the execution of the clustering algorithm, a progress bar is shown. After completion, a dialog window is created that illustrates the clusterings. Figure 6.4 shows this dialog window.

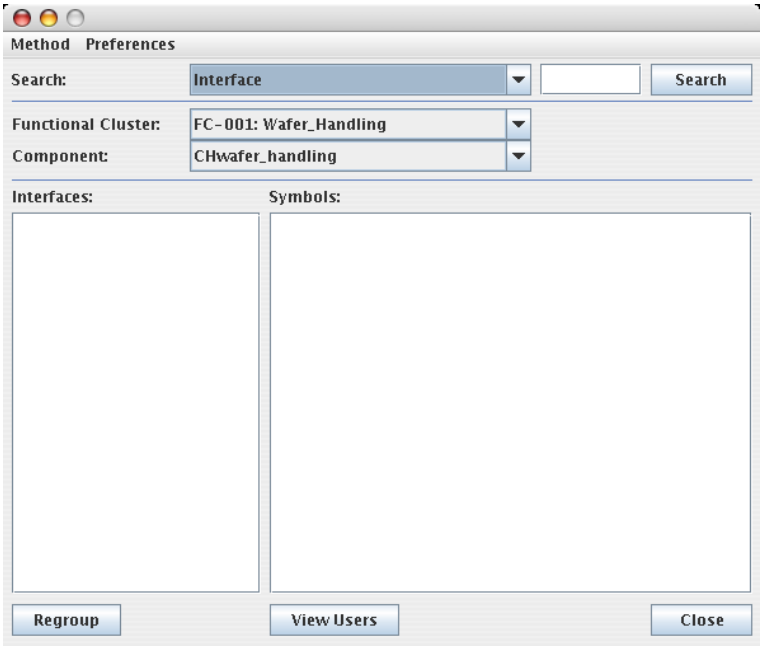

Figure 6.1: The main IRW window.

Note that the title of the dialog window presents the interface/component being regrouped, and the selected distance measure.

In the clusterings dialog window we notice a list of clusterings, where each entry is represented by two numbers. The number on the left illustrates the amount of clusters, and the number on the right illustrates the distance of the next cluster to be merge as a promillage. For instance, 1000 indicates the distance between two closest clusters in the current clustering. Next to the list of clusterings there is a tabbed pane with two tabs. One tab is used for showing the chart of variance-connectivity values of the calculated clusterings and another is used for the presentation of the user distribution of the selected clustering, i.e., how many percent of users uses $1, 2, .., n$ clusters. A dialog window with the user's distribution tab activated is illustrated in Figure 6.5. Furthermore, we are able to view the clusters in the selected clustering in a textual form by clicking the View button. By clicking the View button we create a result dialog window showing a summary of calculated IMAS compliancy criteria, and the clusters' structures of the selected clustering. Each cluster in the result dialog window is accompanied by its level, e.g., functional cluster or building block. Figure 6.6 illustrates such a result dialog window. When we wish to view the concept lattice, we can click the Export button to create a file representing a concept lattice that can be read and visualized by Concept Explorer [3].

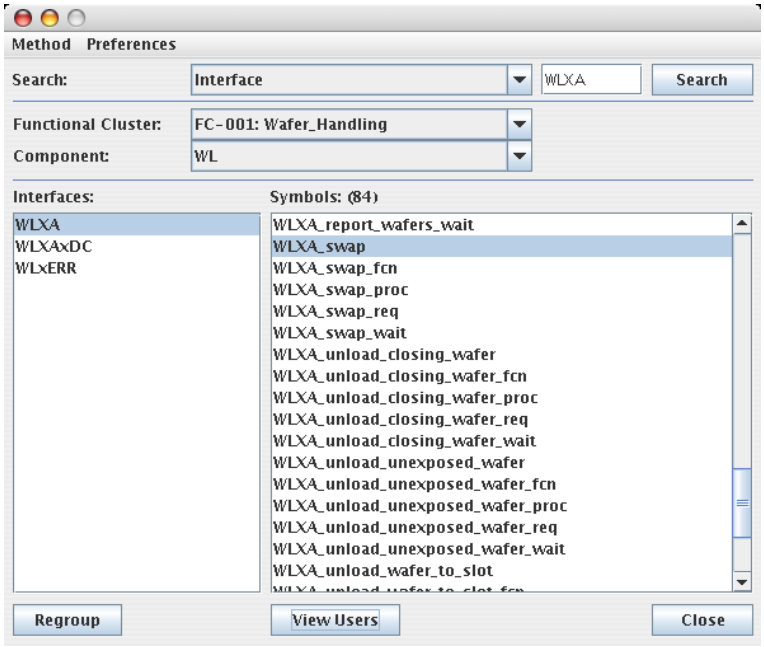

Figure 6.2: The main IRW window loaded with symbols.

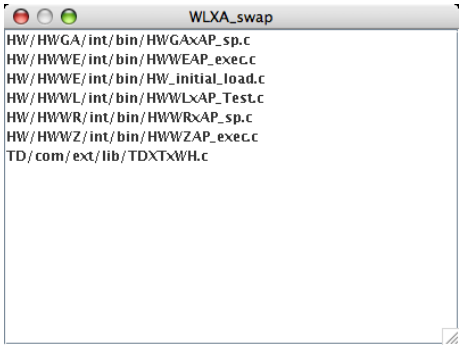

Figure 6.3: The users window of WLXA_swap.

6.2 Problems

The problems we encountered are related with the dependency data extracted by CScout. Here, C's structures and enumerations require special attention, i.e., they are a special kind of symbols.

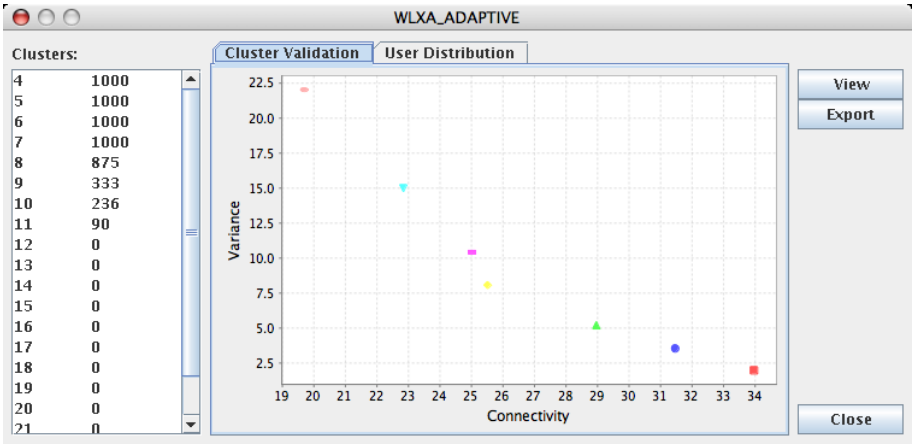

Figure 6.4: The clusterings window (1).

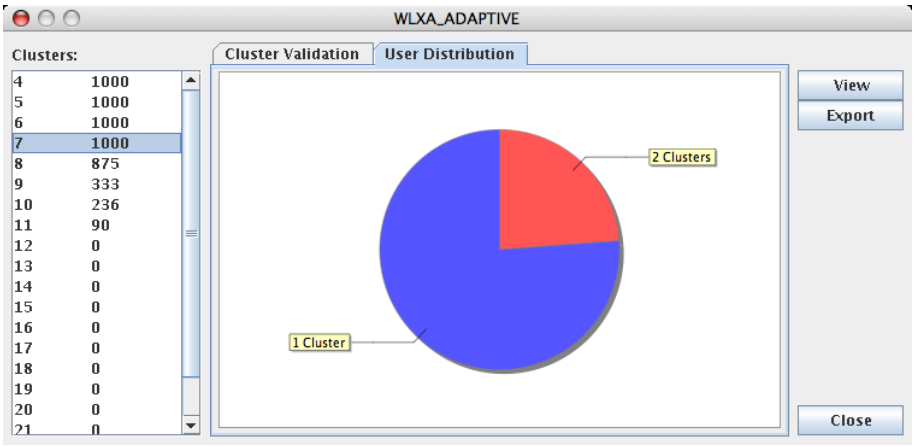

Figure 6.5: The clusterings window (2).

6.2.1 Structure

The first problem is concerned with the elements of a structure (struct). Struct is a keyword that is used in C for declaring a new data-type. Usually, a struct keyword is used to group symbols (variables) together. So, a struct element should always be linked to its container. It is sometimes the case that two symbols with identical names are located in two different structs. In this case, these two symbols are distinct. CScout extracts dependency data through querying the database. Although, in the database, these two symbols are distinct, the data extracted by CScout do not take this distinction in mind. This is due to the fact that symbols in the extracted data are presented as a single independent entity, i.e., we cannot identify the container of a struct symbol. Therefore, two symbols with the same name in

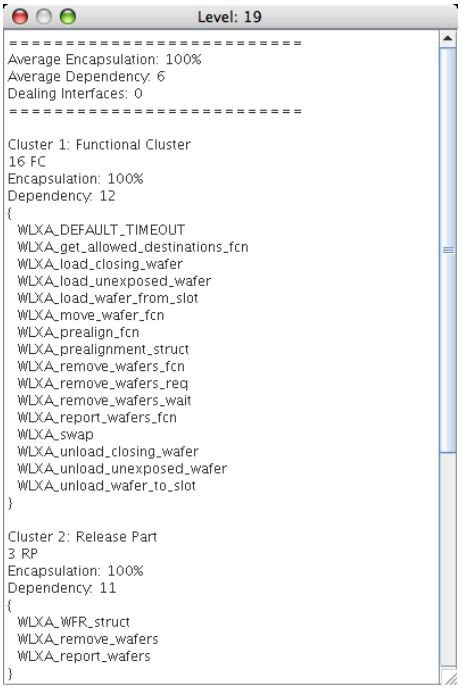

Figure 6.6: The result window.

the same interface will be regarded as the same symbol.

However, we are able to detect whether a symbol is an element of a struct. All struct's elements always starts with a dot in their names. Suppose there is a symbol called `.type`, then we immediately detect that `.type` is a symbol contained in a struct. Nevertheless, we do not know which struct it is contained in. Therefore, we have chosen to provide an option to exclude the symbols that are struct's elements. When this option is enabled, struct's elements are not taken into consideration in the regrouping process. When an architect decides to actually perform the regrouping, the struct should be kept intact.

6.2.2 Enumeration

The next problem is concerned with the elements of an enumeration (enum). Similar to the structure problem, enum's symbols are also presented as single entities. In contrast with struct's elements, enum's elements names always start with the enum's name and thus each enum element is unique. Furthermore, enum's symbols differ from struct symbols in that we cannot detect whether a symbol corresponds to an enum. After a discussion with the architects at ASML, we have come to conclusion that it is in some cases not strictly necessary to group enum's symbols together. The problem presented by enums might benefit the architects, e.g., when enum's symbols are not grouped together by the IRW, this indicates that the enum could possibly be separated. For this reason, we have decided to leave enum elements as they are.

6.2.3 Local User

The inclusion of the local users, i.e., users in the same component as the interface being regrouped, may result in misleading regrouping. For instance, two symbols that are used by two different external users, i.e., users not in the same component as the interface being regrouped, are grouped together due to their local users being common. However, it is often the case at ASML that the external users outweigh the local users, e.g., two symbols having two distinct external users sets should always be separated regardless of their local users. Therefore, we have provided an option that enables and disables the inclusion of local users for the regrouping process.

6.2.4 Tunnelling

At the moment some users uses a symbol through a tunnel, i.e., a special user used for tunnelling purposes. Figure 6.7 illustrates the tunnelling phenomenon. CScout is only able to recognize direct users. Therefore, users `x` and `y` are not recognized as users of symbol `a`. As a consequence, symbols are grouped together because they are used by a tunnel user, when in fact they are used by distinct users. Although the tunnelling phenomenon is used at ASML, it is recognized as an anti-pattern[1] and therefore, it is not preferred. We have decided to ignore this problem. The benefit of ignoring the tunnelling phenomenon is that the IRW is able to detect the anti-pattern, i.e., the generated strange groupings might be caused by the tunnelling phenomenon. Subsequently, the corresponding architect may decide to remove the anti-pattern.

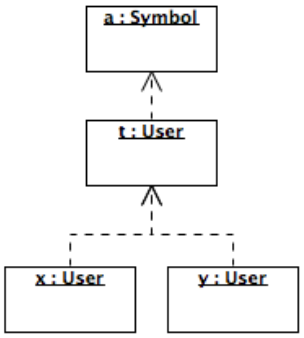

Figure 6.7: Symbol usage through a tunnel.

6.3 Case Study

We are now ready to discuss a case study. Averagely, an interface contains approximately 113 symbols. We have selected an above average interface as a case study, i.e., `WHXA` con-

[1] Anti-patterns comprise the study or specific repeated practices that appear initially to be beneficial, but ultimately result in bad consequences that outweigh the hoped-for advantages.

taining 544 symbols. This interface will be regrouped into several smaller interfaces. We have chosen for the adaptive-link method. We also decided to exclude both struct elements and local users. For the purpose of cluster validation, we calculate the variance and connectivity value for each clustering. The connectivity value is calculated by setting the numbers of contributing neighbours L to 10.

We first discuss the variance-connectivity (cluster validation) chart. Subsequently, after zooming into the chart we select a clustering that is optimal. Furthermore, we discuss the user's distribution corresponding to the selected clustering. We also present the corresponding concept lattice. Finally, we discuss the IMAS compliancy of the result.

The results presented in this section is regrouped by the IRW using the adaptive-link method. We have also regrouped the `WHXA` interface using other methods, which is presented in Appendix C.

6.3.1 WHXA

`WHXA` is an interface of the component WH in the functional cluster FC-001:Wafer_Handling. Currently, for `WHXA` two of the IMAS compliancy criteria have a critical value, i.e., dependency and encapsulation, one has a high value, i.e., change frequency, and one has a medium value, i.e., direct dealing (see Figure 2.6). The dependency criteria is critical because 4443 files are rebuilt per change, which 3.5 times larger than the average. These 4443 files consist among others of 208 .h files, 109 .c files and 14 .ci files (total of 331 files). The encapsulation criteria is critical because only 28% of symbols exported by `WHXA` are on relase part:shared level. In this case study we attempt to regroup `WHXA` such that the dependency and encapsulation criteria are not critical and the dealing criterion is not medium anymore. The change frequency criterion, however, cannot be optimized by the IRW.

When the elements of structs are excluded, `WHXA` contains a total of 544 symbols. From these 544 symbols, 356 symbols are regrouped. The other 188 symbols are not included because they are only used by local users, i.e., users from the component `WH`. The IRW requests for user's confirmation when two clusters having different interface usage levels are to be merged. Excluding the time needed for this confirmation, the IRW requires approximately 30 seconds to complete its execution.

Result

When the regrouping is finished, a variance-connectivity chart is shown. Figure 6.8 illustrates the chart corresponding to the clusterings of the symbols in `WHXA`. It is difficult to draw any conclusion using the chart shown in Figure 6.8. Therefore, we zoom into the rectangle area of the variance-connectivity chart. The zoomed version of the variance-connectivity chart is presented in Figure 6.9.

Looking at Figure 6.9, we can easily detect the optimal clustering. By hovering the mouse above the point of interest in the IRW, we retrieve information about the clustering, and its exact variance and connectivity values. We detect a 'knee' on the clustering with 22 clusters. Here, the variance value dropped significantly while the connectivity value

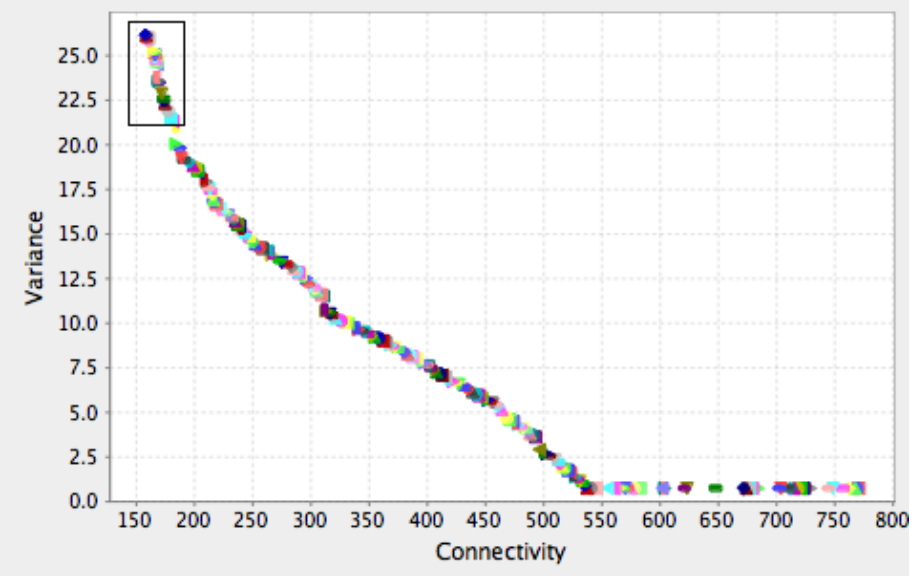

Figure 6.8: WHXA variance-connectivity chart.

remains the same as the next clustering. For this reason, we select a clustering with 22 clusters as the optimal clustering.

Once we have selected the preferred clustering, we can activate the User Distribution tab to view the user distribution in this clustering. Figure 6.10 presents a pie chart of the user distribution of the selected clustering, i.e., the clustering with 22 clusters. From this pie chart, we observe that 59% of all users requires exactly one cluster, and 31% of all users requires exactly two clusters. In this clustering, the worst case scenario, where a user requires all clusters, does not occur. In contrary, the maximum amount of clusters used by a single user is 6. To be more exact, there exists less than 1% of all users that uses more than 6 clusters.

The best option for viewing the clustering and the corresponding relationships between clusters is a concept lattice. The concept lattice for a clustering with 22 clusters is shown in Figure 6.11. The concept lattice illustrates the relation between the clusters, i.e., which clusters share a set of users. Here, every blue node indicates a node containing a cluster and users, and every white node indicates a node containing only users. We omit the users in Figure 6.11 to increase readability.

For the regrouping of `WHXA` interface, the IRW uses approximately 61 megabytes of memory.

IMAS compliancy

The average encapsulation value of all clusters in this clustering is 99%. The worst case in this clustering is a cluster having an encapsulation value of 95%, which indicates that the IMAS compliancy criteria for encapsulation is met. The average dependency value of all

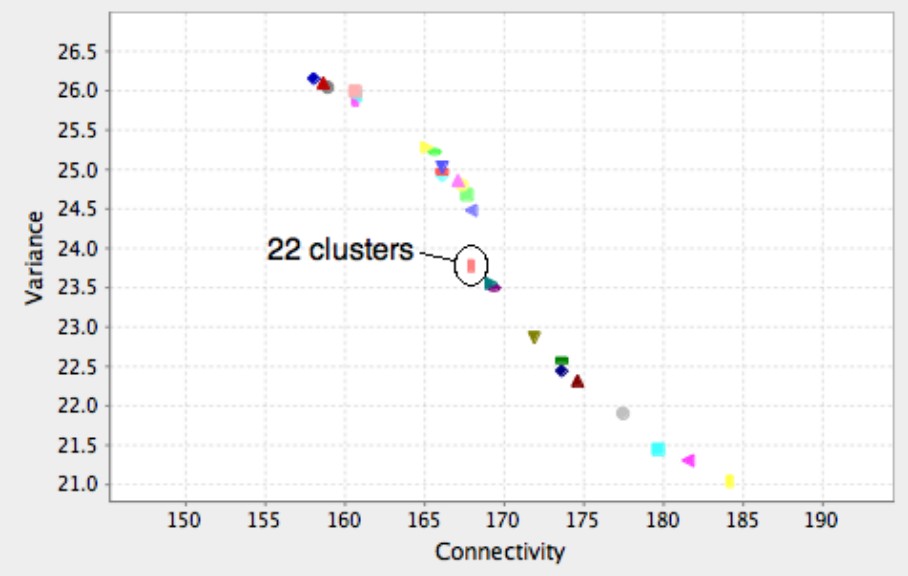

Figure 6.9: WHXA variance-connectivity chart (zoomed).

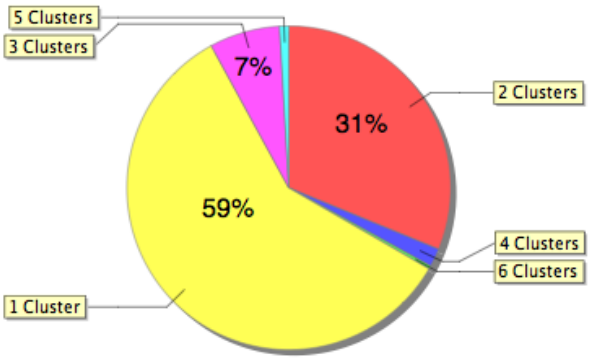

Figure 6.10: WHXA user distribution.

clusters in this clustering is 23. The worst case in this clustering is a cluster having a dependency value of 231, which is less than the 331 files in the original interface. Furthermore, 20 clusters in this clustering are not shared, which indicates that the dealing criteria of the IMAS compliancy is also met for these 20 clusters. The other two clusters consists only of symbols that are shared at respectively the functional cluster and release part level. IMAS has stated that dealing is allowed by interfaces that are specifically built for the dealing purposes, which is the case for the two shared clusters in this clustering.

According to the results, the clustering with 22 clusters contains regroupings of interface

WHXA into 22 smaller interfaces that are IMAS compliant. In other words, the IRW has successfully regrouped the WHXA interface.

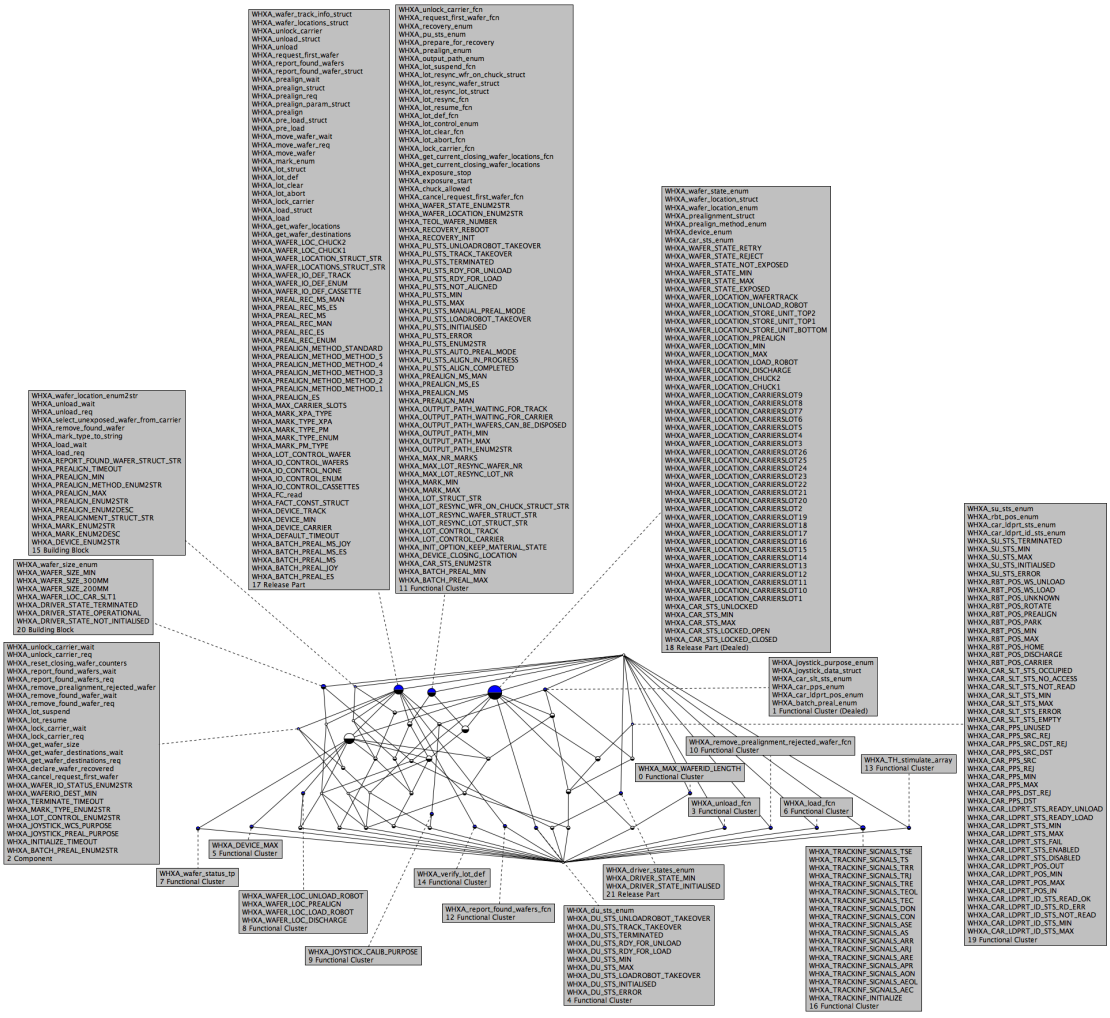

Figure 6.11: WHXA concept lattice.

Chapter 7

Conclusions and Future Work

This chapter provides an overview of the project's contributions in Section 7.1. After this overview, we will reflect on the results and draw some conclusions in Section 7.2. Finally in Section 7.3, some ideas for future work will be discussed.

7.1 Contributions

The main contribution of our research project is the delivery of the IRW, which is a tool that is able to semi-automatically regroup the interface definitions of an interface into multiple optimal interfaces. The IRW has been successfully applied for the regrouping of the interface definitions (symbols) at ASML. These interfaces in the recommended regrouping satisfy a set of IMAS compliancy criteria that are set by ASML in order to determine correct interfaces.

Furthermore, we have investigated a couple of techniques that are able to regroup interface definitions, such as formal concept analysis and cluster analysis. From these techniques we weighed the benefits and the disadvantages. As a result, we have come to conclusion that the hierarchical clustering is the most suitable technique.

Hierarchical clustering requires a distance measure for its execution. We have presented a distance measure that is able to measure the distance between two interface definitions. Additionally, we have devised a method that is able to measure the distance between two clusters called adaptive-link. The adaptive-link method is actually a combination of two already existing methods, i.e., average-link and median-link method.

7.2 Conclusions

In this thesis we have described the IRW. Initially, we generally described ASML and a main product it delivers, i.e., the TwinScan wafer scanner. We have discussed the software architecture of the TwinScan software (control) systems in general and the software interfaces in particular. A brief description of the current situation at ASML is provided illustrating the problems and difficulties of manual regrouping of the interface definitions. Before we actually investigate the possible approaches that tackle the problem, we elicit the requirements

for the IRW from the architects at ASML in the requirements analysis phase. Subsequently, we studied several related works in order to determine the best possible approach for the IRW. After we made our decision of which approach is most suitable, we begin designing and implementing the IRW. Finally, the problems encountered and the results are presented.

We have experienced the performed research project as fast-paced, challenging and innovative. It is fast-paced because we are required to report and discuss the results weekly (agile development). It is challenging because we have to communicate with several architects in order to determine the requirements and to validate the IRW. Furthermore, high expectations are set on the IRW, e.g., the detection of functional coherency. It is innovative because we applied theoretical techniques to a common practical problem of interface restructuring, which hasn't been performed before.

7.3 Future work

Although the IRW is ready for use, there are some points that can be improved. One of them is the detection of struct's and enum's elements. This point, however, is more related to CScout rather than the IRW, i.e., this information should be provided by CScout and processed by the IRW. The use of information about enum's and struct's elements will result in a more sensible grouping of interface definitions (symbols). When an architect notices that an enum is not kept intact, he tend to disapprove the regrouping without even looking at which interface definitions are grouped together. Therefore, the information about enum's and struct's elements is desired for future improvements.

Another point of interest for future addition, is the insertion of functional coherency. The functional coherency can be detected, for instance, by looking at the structure of the code of a function. Two functions with a similar structure or symbols that are dependent should be marked as functional coherent. Consequently, these two functions should always be grouped together. In the future, the detection of functional coherency should be investigated further.

The IMAS compliancy criteria take the change frequency as a criterion for determining correct interfaces. For the purpose of this criterion, ASML administer the change frequency of files. Despite that, the IRW does not take this criterion into consideration. The main reason for this is that the change frequency administered by ASML measures changes of a file and not of a symbol. One possible option is setting the change frequency of a file as the change frequency of all symbols contained by the file, i.e., all symbols get the change frequency of the file. However, this could lead to a misleading grouping. Therefore, in the future, addition of the change frequency of a symbol into the IRW is a noteworthy improvement.

Bibliography

[1] ASML: About ASML - profile. http://www.asml.com/asmldotcom/show.do?ctx=272&rid=362.

[2] ASML: Products - TwinScan. http://www.asml.com/asmldotcom/show.do?ctx=6717.

[3] Concept explorer. http://conexp.sourceforge.net/.

[4] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions*. Dover, New York, 1965.

[5] Rafael C. Andreu and Stuart E. Madnick. A systematic approach to the design of complex systems : application to dbms design and evaluation. Working papers no. 32., Massachusetts Institute of Technology (MIT), Sloan School of Management, April 2003.

[6] Bernd Bruegge and Allen A. Dutoit. *Object-Oriented Software Engineering; Conquering Complex and Changing Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[7] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003. 2003 Jolt Productivity Award Winner.

[8] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *STEP '99: Proceedings of the Software Technology and Engineering Practice*, page 73, Washington, DC, USA, 1999. IEEE Computer Society.

[9] B. Everitt. *Cluster Analysis*. Heineman Educational Books, London, 1974.

[10] Doug Fisher, Ling Xu, James R. Carnes, Yoram Reich, Steven J. Fenves, Jason Chen, Richard Shiavi, Gautam Biswas, and Jerry Weinberg. Applying ai clustering to engineering tasks. *IEEE Expert: Intelligent Systems and Their Applications*, 8(6):51–60, 1993.

[11] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., Secaucus, NY, USA, 1997.

[12] Marco Glorie. Philips medical archive splitting. Master's thesis, Delft University of Technology, 2007.

[13] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[14] Julia Handl, Joshua Knowles, and Douglas B. Kell. Computational cluster validation in post-genomic data analysis. *Bioinformatics*, 21(15):3201–3212, 2005.

[15] Anil Hemrajani. *Agile Java Development with Spring, Hibernate and Eclipse (Developer's Library)*. Sams, Indianapolis, IN, USA, 2006.

[16] Jeffrey A. Hoffer, Joey F. George, and Joseph S. Valacich. *Modern Systems Analysis and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[17] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[18] Jr. Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, March 1963.

[19] Stephen C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, September 1967.

[20] Chung-Horng Lung, Xia Xu, Marzia Zaman, and Anand Srinivasan. Program restructuring using clustering techniques. *J. Syst. Softw.*, 79(9):1261–1279, 2006.

[21] Chung-Horng Lung, Marzia Zaman, and Amit Nandi. Applications of clustering techniques to software partitioning, recovery and restructuring. *J. Syst. Softw.*, 73(2):227–244, 2004.

[22] J.B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967.

[23] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–59, Washington, DC, USA, 1999. IEEE Computer Society.

[24] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 45, Washington, DC, USA, 1998. IEEE Computer Society.

[25] Clark F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Comput.*, 21(8):1313–1325, 1995.

[26] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, 1997.

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

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

[29] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–110, New York, NY, USA, 1998. ACM Press.

[30] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques, 2000.

[31] Thomas Tilley, Richard Cole, Peter Becker, and Peter Eklund. A survey of formal concept analysis support for software engineering activities. *Lecture Notes in Computer Science*, 3626/2005:250–271, July 2005.

[32] Paolo Tonella. Concept analysis for module restructuring. *IEEE Trans. Softw. Eng.*, 27(4):351–363, 2001.

[33] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, page 122, Washington, DC, USA, 2004. IEEE Computer Society.

[34] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 246–255, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[35] R. Wille. *Restructuring lattice theory: an approach based on hierarchies of concepts.* Ordered sets, Reidel, Dordrecht-Boston, 1982.

[36] Karl Erich Wolff. A first course in formal concept analysis: How to understand line diagrams. *SoftStat '93: Advances in Statistical Software*, 51(4):429–438, 1995.

[37] L. A. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets Syst.*, 100(supp.):9–34, 1999.

Appendix A

Glossary

Building Block (BB): A collection of components.

Cluster: A group of entities.

Cluster Analysis An analysis technique that is able to detect clusters.

Component (CC): A collection of files and interfaces.

Concept Lattice: A graph illustrating a formal context.

CScout: An analysis tool that is able to extract symbol-user dependencies.

Dealing: A phenomenon where an interface shares the symbols of other interfaces, either directly or indirectly.

Formal Concept: A pair of objects and attributes, such that all attributes are valid for all objects in the concept.

Formal Concept Analysis: An analysis technique based on formal contexts and formal concepts.

Formal Context: A mathematical structure that is used to describe formally the relations between objects and attributes.

Functional Cluster (FC): A collection of functional coherent building blocks.

Hierarchical Clustering A cluster analysis based on incremental generation of clusters.

IDT: Interface Dependency Tree.

IMAS: Interface Management and Archive Structure.

IMAS compliancy criteria: A set of criteria set by the IMAS project to determine good interfaces.

Interface (I/F): A collection files defining a set of symbls.

Interface Provider: A release part, a functional cluster or a building block.

IRW: Interface Regroup Wizard.

MLOC: Million Lines of Code.

Release Part (RP): A collection of building blocks required by a certain release.

Shared Interface: An interface where at least one symbol is used by other interfaces.

Sharing: See: Dealing.

Symbol: A representation of a macro definition, a type definition or a function declaration.

User: A file that uses a symbol.

Appendix B

Splitting interfaces

The guideline presented below is obtained from ASML's IMAS documentation. It illustrates the manual method for splitting interfaces.

Make sure an activity is set (work on)

1. Make sure you have the proper components in your buildscope.
   Build on `/vobs/litho` level. This to make sure that everything is generated and building BEFORE you begin.
2. Create the new .ddf files with the new interfaces. Found in e.g., `machine_control_and_lo/KS/com/ext/ddf`.
3. Adapt the makefile so that the new .ddf are used and compiled to the appropriate result files. Found in e.g. `machine_control_and_lo/KS/com/ext/inc`.
4. Create new include files which include the `met.h` and `typ.h` files as found in e.g. `machine_control_and_lo/KS/com/ext/inc`.
5. Create new `_isim.c` files when necessary. Also update the makefile. Found e.g. in `machine_control_and_lo/KS/com/ext/lib`.
6. Perform qac check on the `_sim.c` files as these are new files for QAC you met get problems when these were already present in the original file, e.g., `ccmake -c qac -f KSXAxDC.c`.
7. Make sure the scope files are updated by the FC architect. Do not forget to run `ccupdate_scope` so that the proper links in the `xinc` directories are made.
8. You can test if everything is OK by building the component.
9. Create a new workinstruction. This way the changes of the global search and replace can be tracked more easily and potential problems can be undone. Do not forget set it active.
   The following steps will make a rebase action difficult. So make sure the search

and replace action is done as late as possible and that a rebase is done before it is performed.

10. Rename the functions etc in the other components.

    - Create a file like:
      `/sft/teamsdoc/ms_ls_team/tools/input_files/KSXAxDC.def`
      in an appropriate place. This file contains the names which must be renamed.
    - Goto the module like metro (`/vobs/litho/metro`) or for example first to KV (`/vobs/litho/metro/KV/`)
    - Run tool e.g. `/sft/teamsdoc/ms_ls_team/tools/split_tool.py /sft/teamsdoc/ms_ls_team/tools/input_files/KSXAxDC.def KSXAxDC y`
      (this means everything mentioned in `KSXAxDC.def` will renamed to the new prefix as mentioned on the command line. y means that it will be checked out out ClearCase).
      Note that you have to update the #includes by hand.

11. Update the makefile of the components which are updated. I.e. the old library must be replaced by 1 or more new libraries.

12. Update the content of the xlib directories of the components which are updated. I.e. the link to the old library must be replaced by 1 or more new libraries (same as in the makefile).

    - Check the `xlib` directory out in ClearCase, e.g.,
      `cd /vobs/litho/metro/xbin` and then `cleartool co -nc`.
    - Remove the old link `cleartool rmname libKSXA.so`
      Tip: do a `ls -l libKSXA.so` first so the full path can be used in the next step.
    - Add the new link.
      `cleartool ln -s ../../xlib/libKSXAxSWAP.so`.
      Links need to be made in several places, e.g.,
      `/vobs/litho/xlib, /vobs/litho/metro/xlib`

13. Execute `ccmake libref` to verify that the proper libraries are added / removed

14. Update the `PKG_contents` file so that it includes the new .so and .ddb files. Found in e.g. `machine_control_and_lo/KS/com/pkg`

15. `ccmake` and fix the problems. Do not forget to also do a `ccmake` on `/vobs/litho` level in the end.

16. Check everything in in ClearCase.

Appendix C

Results

This appendix presents the results produced by the IRW using the other methods, i.e., single-link, complete-link, average-link and median-link. We notice that the single-link method attempt to minimize the connectivity while disregarding the variance. In contrast to single-link, the complete-link method attempt to minimize variance while disregarding connectivity. The average-link and median-link method are compromises of both methods.

C.1 Single

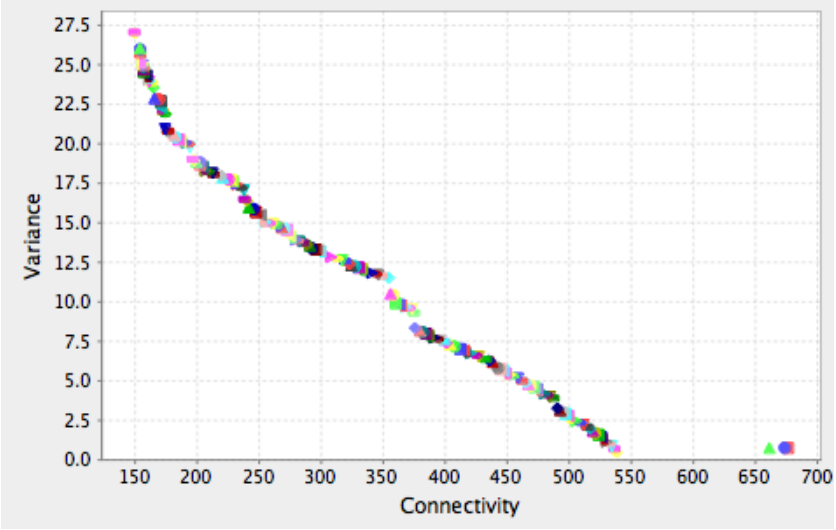

Figure C.1: WHXA variance-connectivity chart.

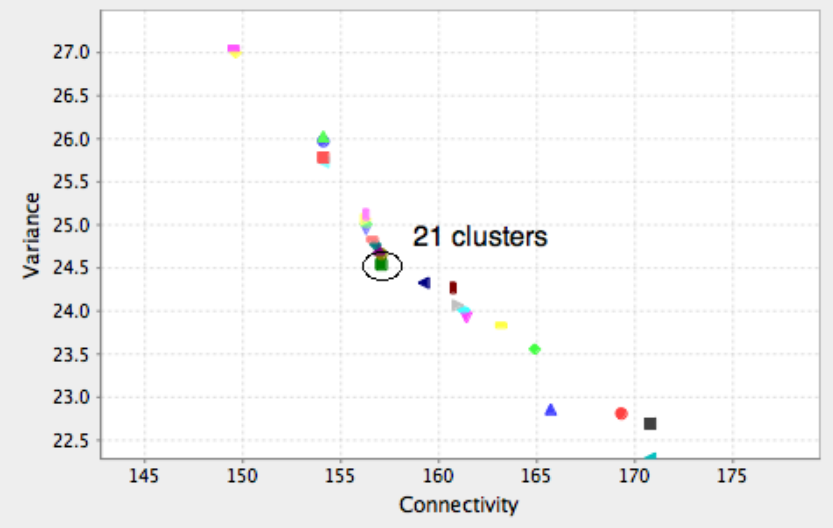

Figure C.2: WHXA variance-connectivity chart (zoomed).

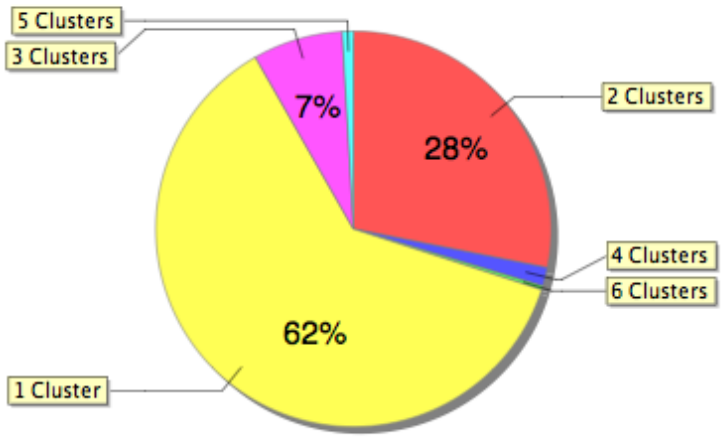

Figure C.3: WHXA user distribution.

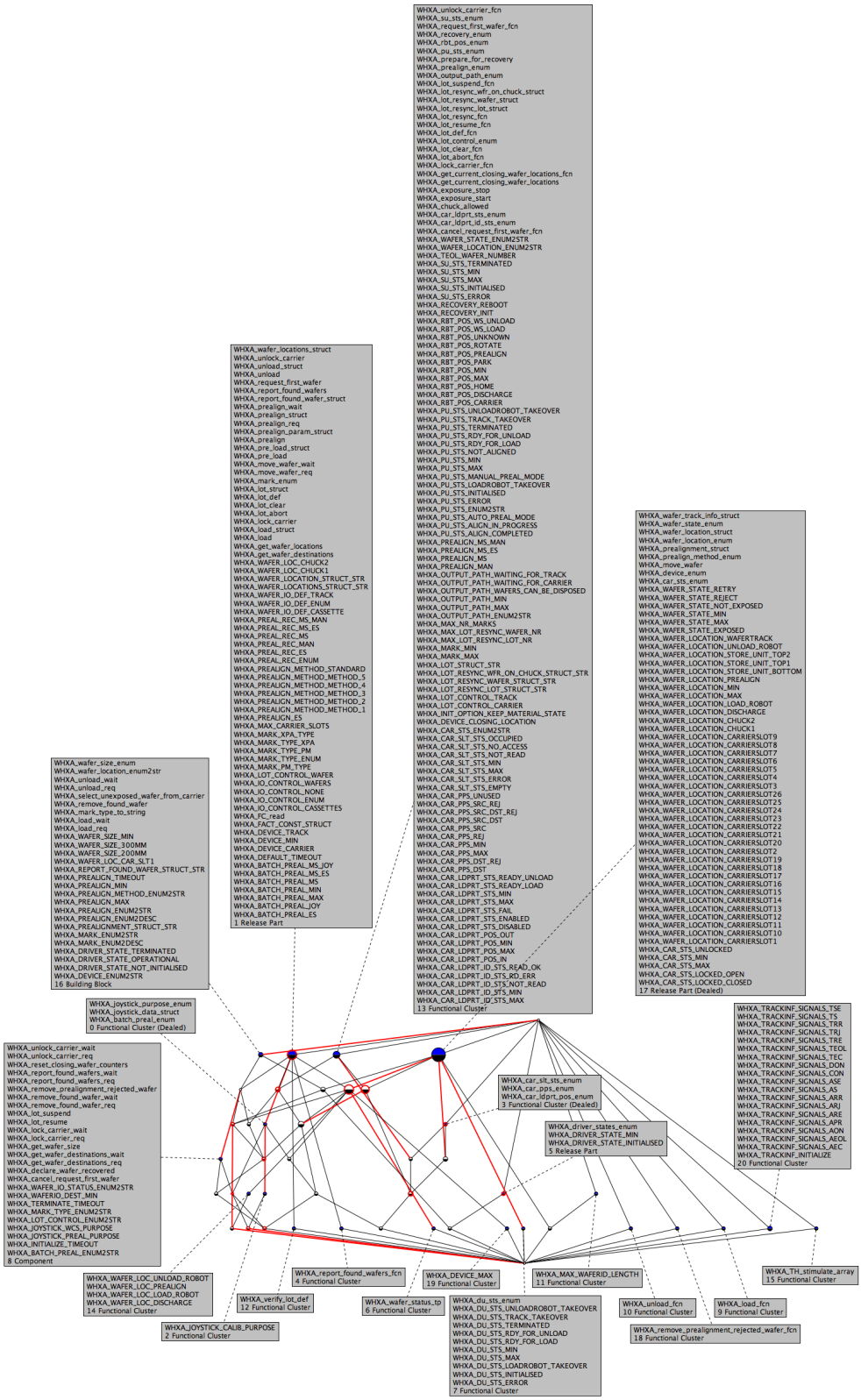

Figure C.4: WHXA concept lattice.

C.2 Complete

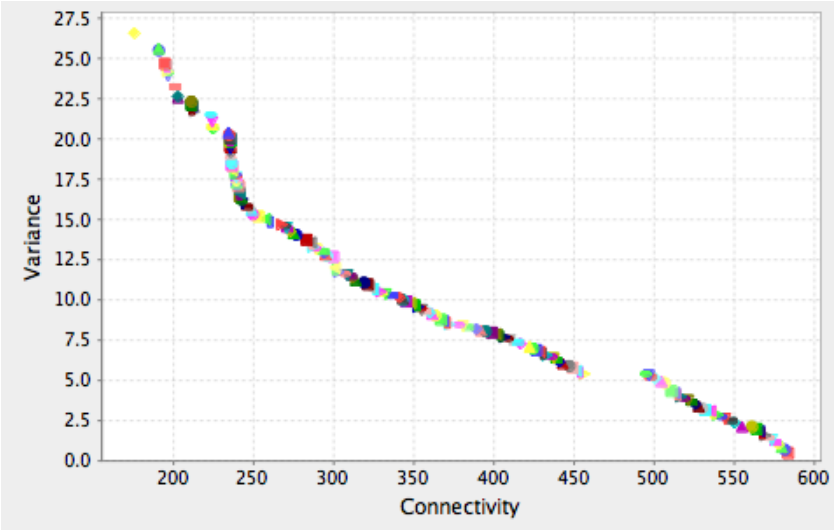

Figure C.5: WHXA variance-connectivity chart.

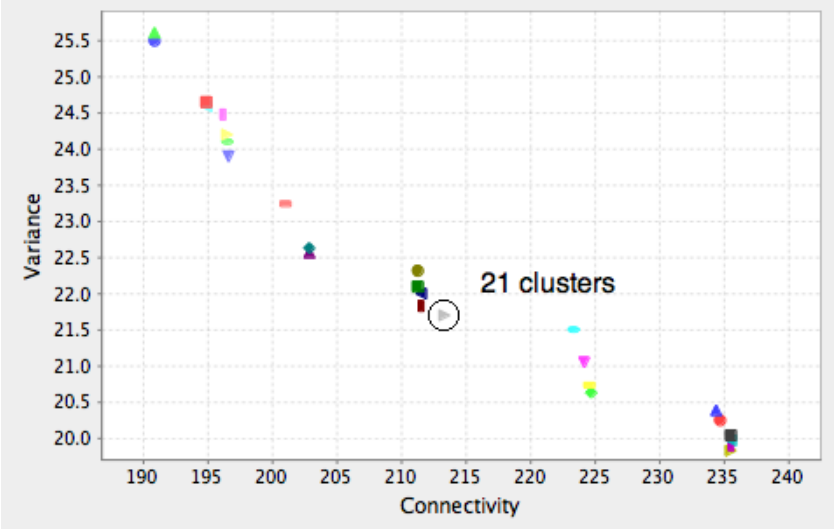

Figure C.6: WHXA variance-connectivity chart (zoomed).

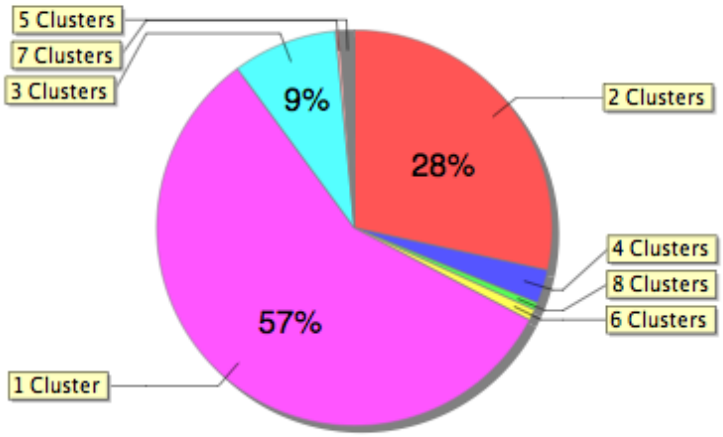

Figure C.7: WHXA user distribution.

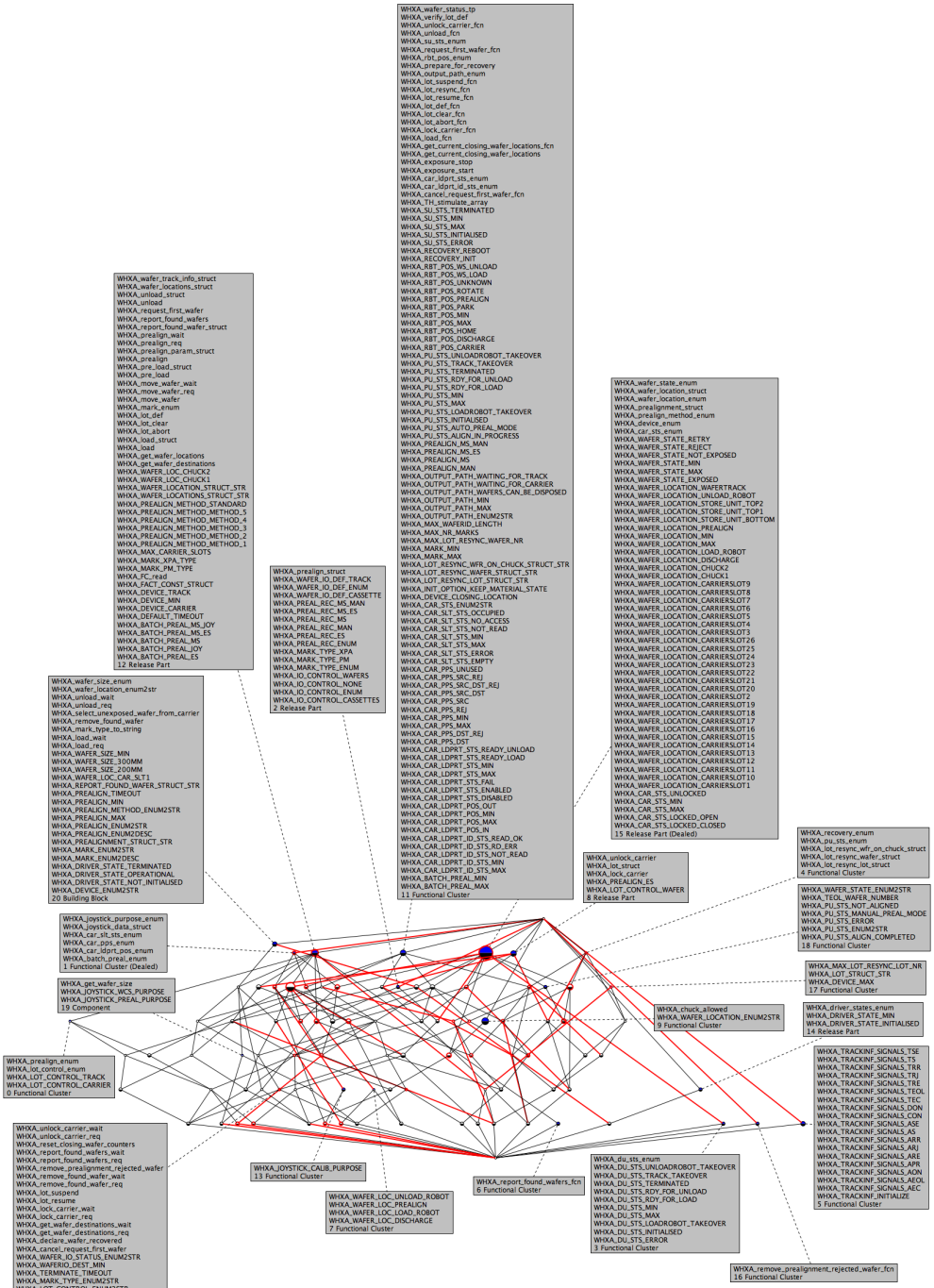

Figure C.8: WHXA concept lattice.

C.3 Average

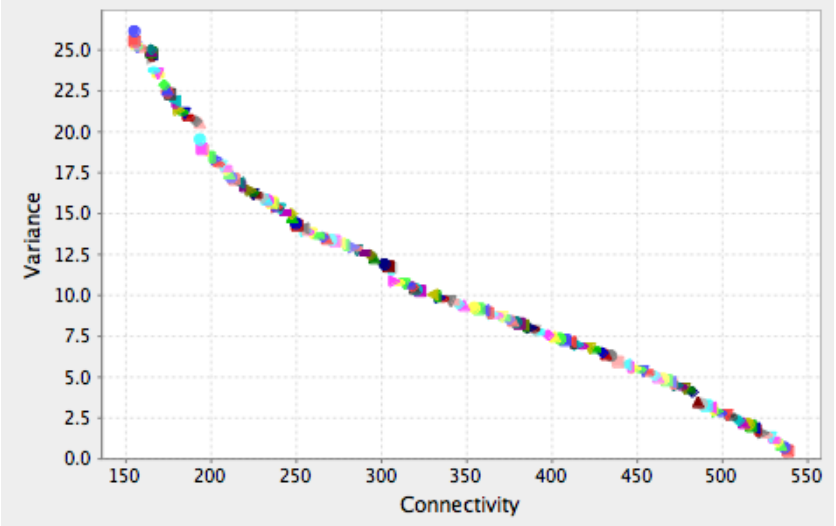

Figure C.9: WHXA variance-connectivity chart.

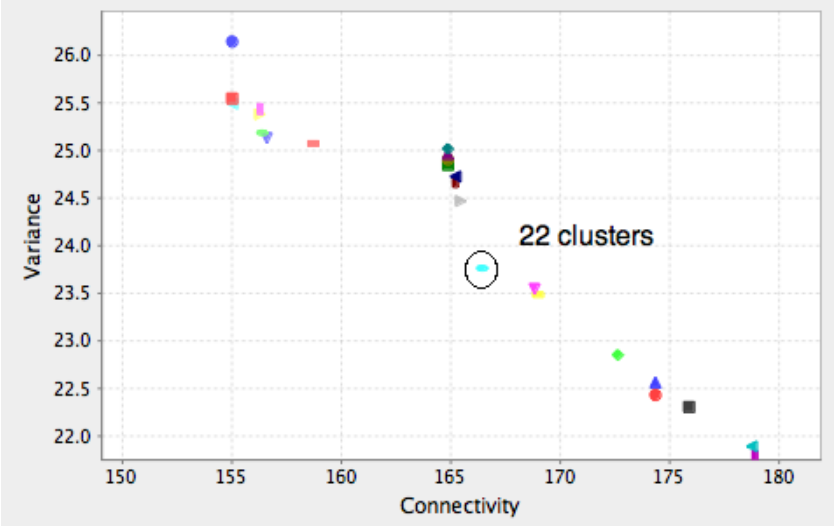

Figure C.10: WHXA variance-connectivity chart (zoomed).

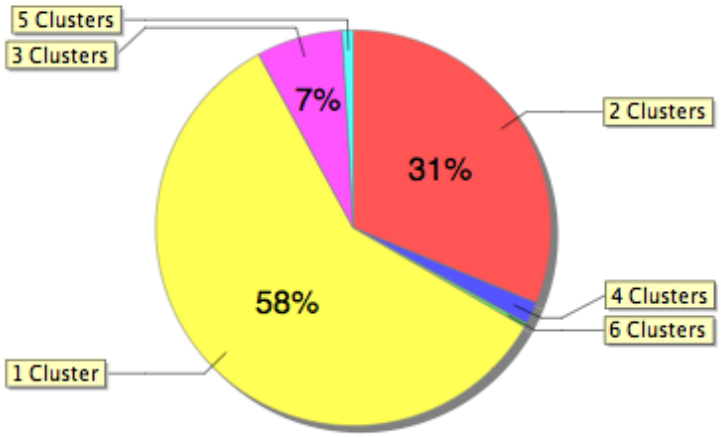

Figure C.11: WHXA user distribution.

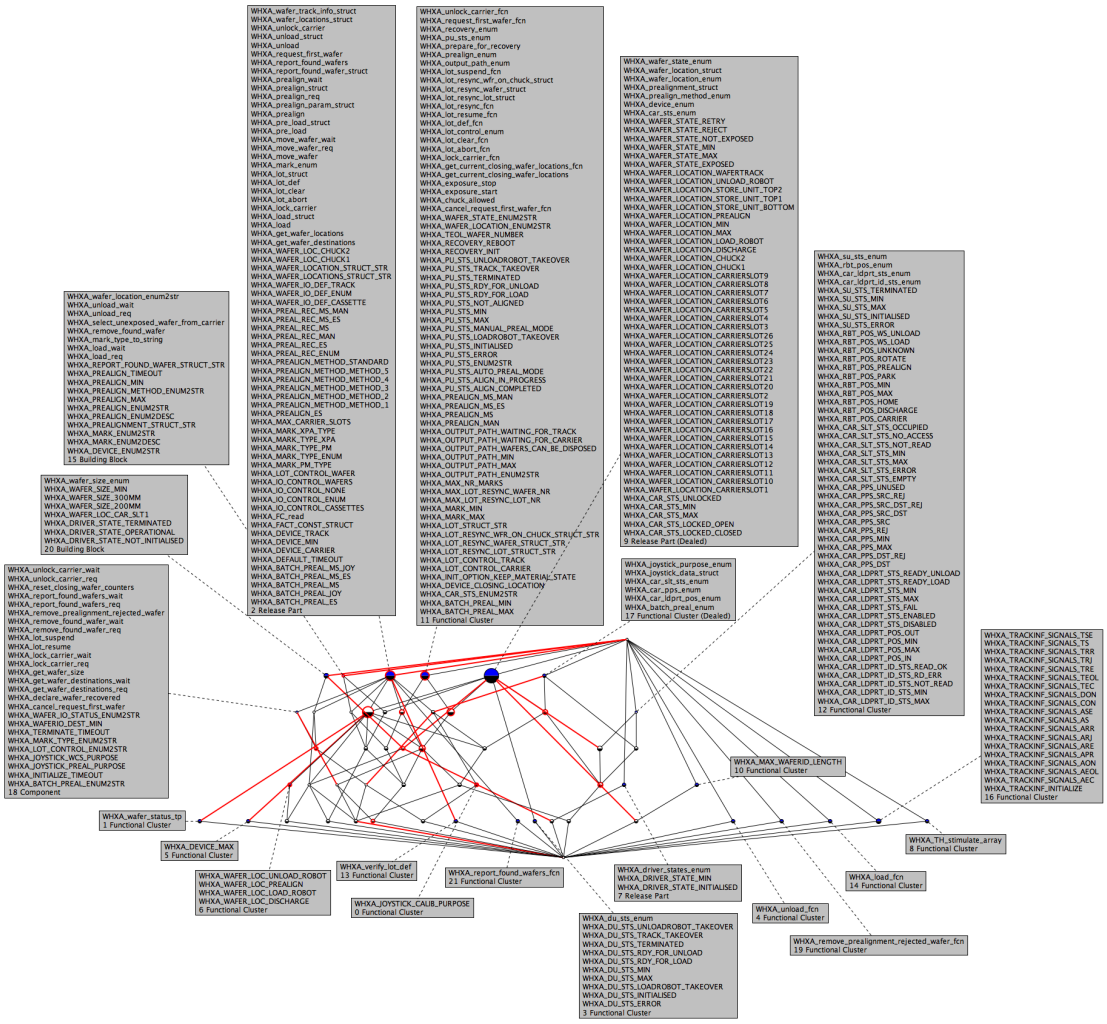

Figure C.12: WHXA concept lattice.

C.4 Median

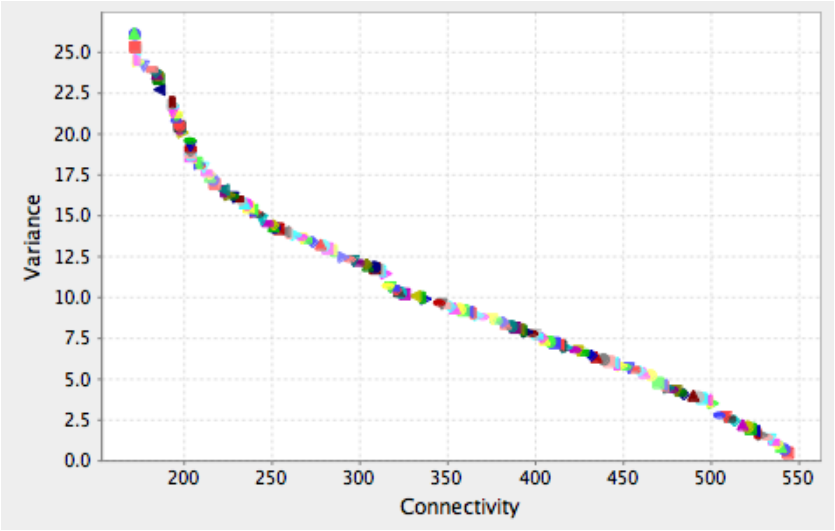

Figure C.13: WHXA variance-connectivity chart.

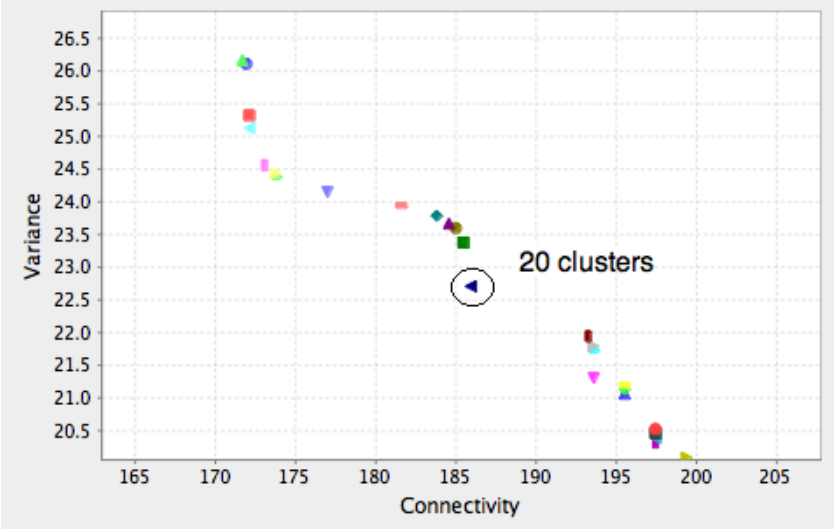

Figure C.14: WHXA variance-connectivity chart (zoomed).

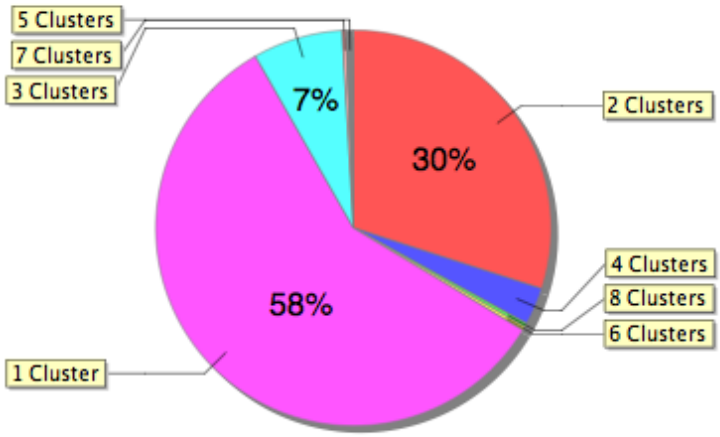

Figure C.15: WHXA user distribution.

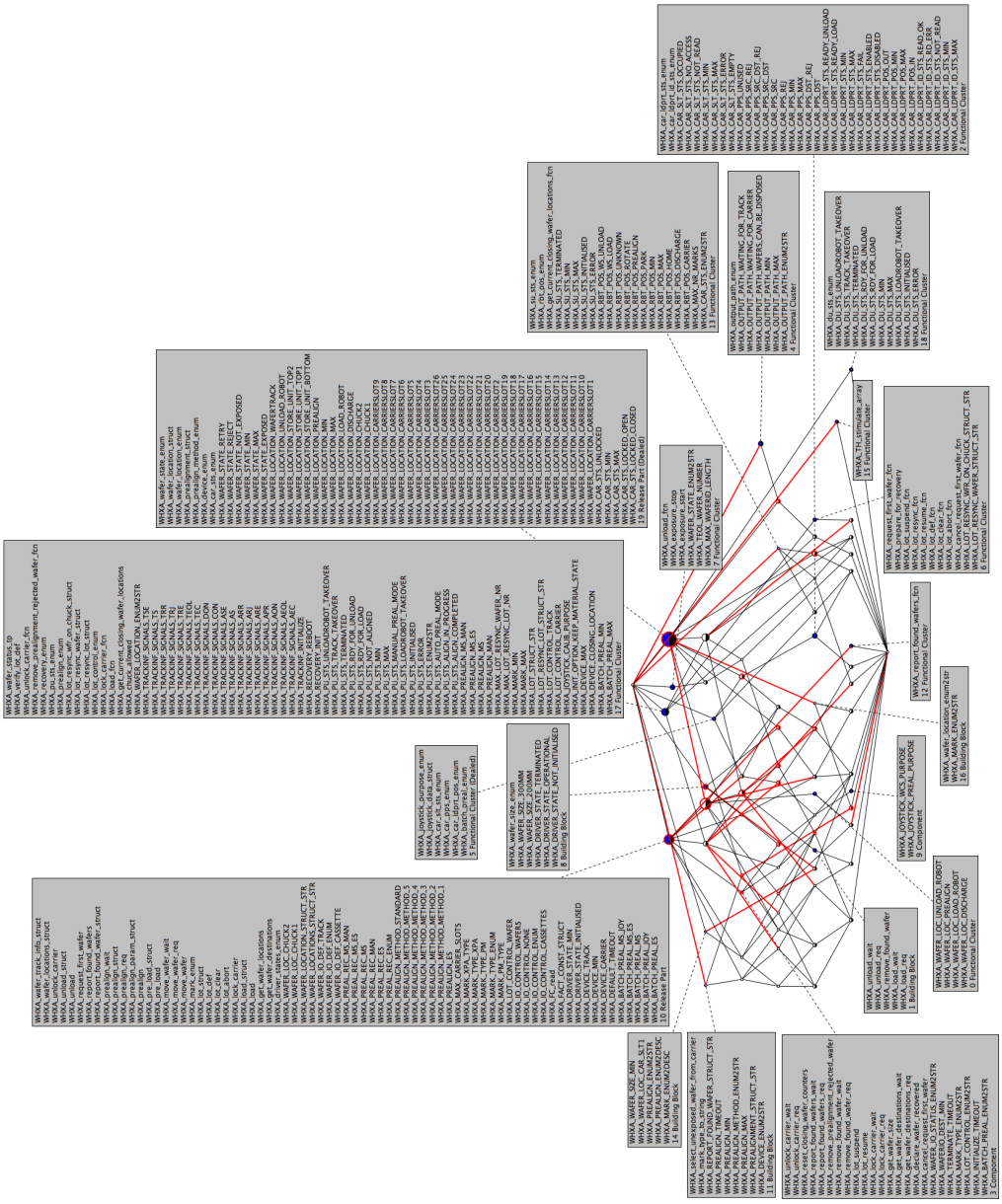

Figure C.16: WHXA concept lattice.