A Survey of Formal Concept Analysis Support for Software Engineering Activities

Thomas Tilley, Richard Cole, Peter Becker and Peter Eklund

School of Information Technology and Electrical Engineering
University of Queensland
Brisbane, Australia
{tilley, rcole, pbecker, peklund}@itee.uq.edu.au

Abstract. Formal Concept Analysis (FCA) has typically been applied in the field of software engineering to support software maintenance and object-oriented class identification tasks. This paper presents a broader overview by describing and classifying academic papers that report the application of FCA to software engineering. The papers are classified using a framework based on the activities defined in the ISO12207 Software Engineering standard. Two alternate classification schemes based on the programming language under analysis and target application size are also discussed. In addition, the authors work to support agile methods and formal specification via FCA is introduced.

1 Introduction

The application of Formal Concept Analysis (FCA) to real world problems has relied on software tools for the both the construction of contexts and the rendering of concept line diagrams. These tools include the DOS-based ConImp[8], Anaconda [1], Toscana[26] and Cernato [2] for the Windows operating system, and more recently the Java-based ToscanaJ [3]. While the Toscana family of tools operate on information stored in relational databases, Cernato presents an interface with the familiar appearance of a spread-sheet application.

In the domain of software engineering FCA has typically been applied to support software maintenance activities — the refactoring or modification of existing code — and to the identification of object-oriented (OO) structures. There is also a body of literature reporting the application of FCA to the identification and maintenance of class hierarchies in database schemata[46,45,13]. While a database system typically forms the backbone of most Computer Assisted Software Engineering (CASE) tools the discussion of database related applications is beyond the scope of this paper.

Beyond the identification of classes, FCA has also been applied to other areas of software engineering including requirements analysis and component retrieval. The aim of this paper is to provide a broad overview of the area by describing and classifying

[1] See `http://www.mathematik.tu-darmstadt.de/~vogt/software-engl.html#FCAL`
[2] See `http://navicon.com.de/`
[3] See `http://toscanaj.sourceforge.net/`

academic papers that report the application of FCA to a range of software engineering activities. These papers are classified using a framework based on the activities defined in the ISO12207 Software Engineering standard[29]. Two alternate classification schemes based on the programming language applicability and target application size are also presented along with a brief analysis of authorship and citation patterns within the survey literature.

The next section of the paper introduces the framework used to classify the different reported approaches based on their applicability to well-defined software engineering activities. Section 3 then presents the classified papers to provide an overview of FCA support for software engineering. Approaches related to the early phases of software development and an application to formal specification are discussed in Section 4. Section 5 reports work related to software maintenance and the authors own work to support Agile Methods is also introduced before Section 6 concludes the paper.

2 A Software Engineering Life Cycle Framework

To understand how software engineering can be supported by FCA we must first have some understanding of what software engineering is or at least the processes involved. This section of the paper sets out a framework that will be used to classify papers from a software engineering development perspective.

The development of software has traditionally been described by life-cycle models. These models grew out of a need to better understand and manage the software engineering process which has been characterised by failed, late, and bug-laden projects. Royce[43] proposed the classic “waterfall” model which consists of seven *steps* or *phases* that proceed in a linear fashion: System Requirements, Software Requirements, Analysis, Program Design, Coding, Testing, and Operations.

The waterfall model focuses heavily on the documentation produced during each implementation phase and there may be some iteration between successive steps. Royce realised that sometimes iterations happen across non-consecutive steps which is undesirable. To address this he proposed some extensions to alleviate the “risk” which largely focused on the production of additional documentation.

The “V” model[42] is a variant of the waterfall model where each step down the left hand side of the “V” has a corresponding validation or verification step on the right hand side. This model presents the opportunity for more “formal” development where documents from the left hand-side feed into the validation activities on the right. The spiral model[4] is another alternative life-cycle that directly incorporates risk analysis as one of four major activities that also includes: planning, engineering and customer evaluation. Starting in the centre of a spiral the developers work through a planning phase, followed by risk analysis, the engineering of a prototype system and then customer evaluation. The cycle then repeats and each move around the spiral progresses outwards towards the final system in an evolutionary fashion.

In addition to these three examples a number of other life-cycle models exist and the most appropriate model to use for a given project may depend on a number of factors including the type of project, the development style and the organisational maturity of both the developers and the customer. An alternative to the classic life-cycle approaches is to use a meta-model that defines common software engineering activities independently of a particular life-cycle model. Developers can then choose the most-appropriate life-cycle for their project and the activities can be mapped onto the chosen model.

2.1 ISO 12207

The ISO 12207 Software Engineering Standard describes such a meta-model for software engineering life-cycle processes that consists of thirteen activities that can be mapped onto a chosen life-cycle model[29]. The first activity, "process implementation", is related to starting the methodology itself, while another four of the activities are system related: "System requirements analysis", "System architectural design", "System integration" and "System qualification". The remaining eight are relate to the software itself and the standard notes that "these activities and tasks may overlap or interact and may be performed iteratively or recursively". From the IEEE Standard Glossary of Software Engineering Terminology[28] the definitions of the software related activities are:

- **requirements analysis** The process of studying user needs to arrive at a definition of system, hardware, or software requirements.
- **architectural design** The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.
- **detailed design** The process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to be implemented.
- **coding and testing** Where *coding* is defined as "…the process of expressing a computer program in a programming language" and *testing* is "the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items".
- **integration** The process of combining software components, hardware components, or both into an overall system.
- **qualification testing** Testing conducted to determine whether a system or component is suitable for operational use.
- **installation** The period of time in the software cycle during which a software product is integrated into its operational environment and tested in this environment to ensure that it performs as required.
- **acceptance support** Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system.

2.2 Software Maintenance

In addition to the eight activities defined above an understanding of software maintenance is also required. The process of software maintenance requires iteration through some or all of the previously defined activities and in terms of the waterfall model it could be thought of as a feedback loop to previous stages. The IEEE Standard Glossary of Software Engineering Terminology defines it as:

– **software maintenance** The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

The next section of the paper uses these nine activities within a framework to classify academic papers reporting the application of FCA to software engineering activities. The intention is not to classify a paper according to a single activity but to record all of the activities that are supported by a reported approach.

3 FCA Support for Software Engineering activities

This section of the paper presents an overview of FCA support for software engineering via a paper survey. The nine software engineering activities defined in Section 2 are used to classify 42 academic papers and theses published between 1992 and 2003. The papers report the application of FCA to software engineering and the approaches they describe are summarised in Sections 4 and 5. Note that this paper assumes some familiarity with FCA and the reader is pointed to Ganter and Wille's classic text for further details[22].

The papers within the survey were analysed using FCA and a many-valued context was constructed that considers the papers as a set of formal objects. Part of the context appears in Table 1. References to papers included in the survey use the naming format adopted by the Research Index (formerly known as "CiteSeer") digital library [4]. Paper names are composed of the first author's surname, the last two digits of the year of publication, and the first word of the title (excluding words like "an", "the", "a", etc.). For example the paper "On The Inference of Configuration Structures from Source Code" published by Krone and Snelting in 1994 appears as *Krone94inference*[30].

A first classification of the papers, derived from the many valued context by considering the ISO activities as scale attributes, appears in Figure 1. From the line diagram it can be seen that 23 of the papers describe applications to both *software maintenance* and *detailed design*. Note that only the object count has been included for this concept to aid the readability of the diagram. These papers are typically reporting the use of FCA to identify class candidates in legacy code or the maintenance of class hierarchies [5]. Considering that concept lattices are hierarchies this is an obvious application. An emerging body of literature related to *requirements analysis* can also be seen although typically a number of the papers are reporting the same example. It is also worth noting that there is only a single paper describing an application to *testing* and no papers explicitly reporting application to *software integration, qualification testing, acceptance support* or *coding*.

[4] See `http://citeseer.nj.nec.com/`

[5] To avoid confusion the terms "class" or "class candidate" will typically be used to refer to Object-oriented objects as opposed to formal objects in FCA.

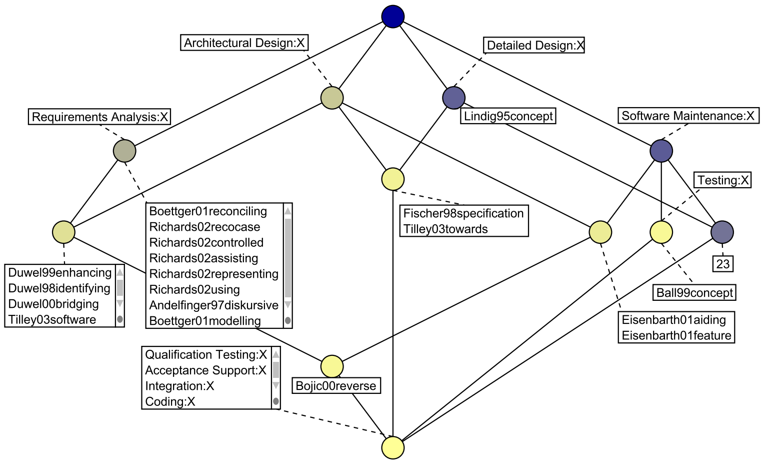

Fig. 1. A formal concept lattice showing the 42 papers categorised according to the ISO software engineering activities they support.

3.1 Alternate Classification Schemes

In addition to the ISO activity categorisation a number of other attributes were used to categorise the papers in the survey. In total 133 attributes were identified including: the names of the authors, citations of other papers in the survey, the year of publication, inputs, outputs, target application languages (e.g. C++, Java) and the "size" of any reported application target.

Target Application Language The context in Table 1 represents the application of the techniques described in a paper to a particular programming or design language. The attributes here are the programming languages: C, C++, COBOL, Fortran, Java, Modula-2, SmallTalk-80, and the design or specification languages: OMT, UML and Z. Both procedural and OO languages are represented. The attribute values record the size of any reported target application in KLOC ("thousand lines of code) – for example, 106 KLOC represents an application containing 106,000 lines of source code. KLOC is also sometimes referred to as SKLOC (Source Thousands Lines of Code) and is a metric that is often reported to indicate project size in software engineering. While KLOC is not necessarily a consistent or meaningful metric it does gives a raw indication of the size of applications to which the technique has been applied and may imply tool support.

A number of papers report application to a specific language but do not report the size of a particular application - the KLOC value for these papers appears as "0" in the context. It is also interesting to note that where a non-zero value repeats in the context it typically refers to the same example being reported in a number of papers.

Figure 2 presents a concept lattice that treats Table 1 as a formal context, ignoring the differences between values. It can be seen that 15 of the papers do not report any application to a particular programming or design language. Also of interest is the paper *Snelting00software*[51] which reports applications to six of the ten languages. This is

	C	C++	COBOL	Fortran	Java	Modula-2	OMT	Smalltalk80	UML	Z
Andelfinger97diskursive [1]										
Ball99concept [2]	0									
Boettger01modelling [6]										
Boettger01reconciling [7]										
Bojic00reverse [5]		0								
Canfora99case [9]			200							
Dekel02applications [12]					0					
Duwel98identifying [15]										
Duwel99enhancing [14]										
Duwel00bridging [16]										
Eisenbarth01aiding [17]	0									
Eisenbarth01feature [18]	76									
Fischer98specification [19]										
Funk95Algorithms [21]	1.6									
Godin93building [23]								0		
Godin95applying [25]										
Godin98design [24]										
Huchard99from [27]					0					
Krone94inference [30]	1.6									
Kuipers00types [31]			100							
Leblanc99environment [32]		0			0		0		0	
Lindig95concept [33]										
Lindig97assessing [34]			5	106		1.5				
Richards02assisting [35]										
Richards02controlled [36]										
Richards02recocase [38]										
Richards02representing [37]										
Richards02using [39]										
Sahouri97applying [44]	47									
Schupp02right [47]		0								
Siff97identifying [48]	28									
Snelting96reengineering [49]	1.6									
Snelting98concept [50]	1.6		0	106		1.5				
Snelting98reengineering [52]		0								
Snelting99reengineering [53]	0				9					
Snelting00software [51]	1.6	0	0	106	9	1.5				
Snelting00understanding [54]		0			12					
Tilley03software [58]										0
Tilley03towards [57]										0
Tonella99object [60]		21								
Tonella01concept [59]	249									
vanDeursen98identifying [61]			100							

Table 1. A Formal Context showing reported application languages for the 42 papers in the survey. The attribute values represent the size of the application in KLOC("thousand lines of code"). A KLOC value of '0' indicates that a paper described application to a particular language but no size was reported.

a paper by Snelting that surveys earlier results from a number of papers he has either authored or co-authored. Another line diagram in Figure 3 summarises the same context but only considers the maximum reported KLOC across all languages for each paper.

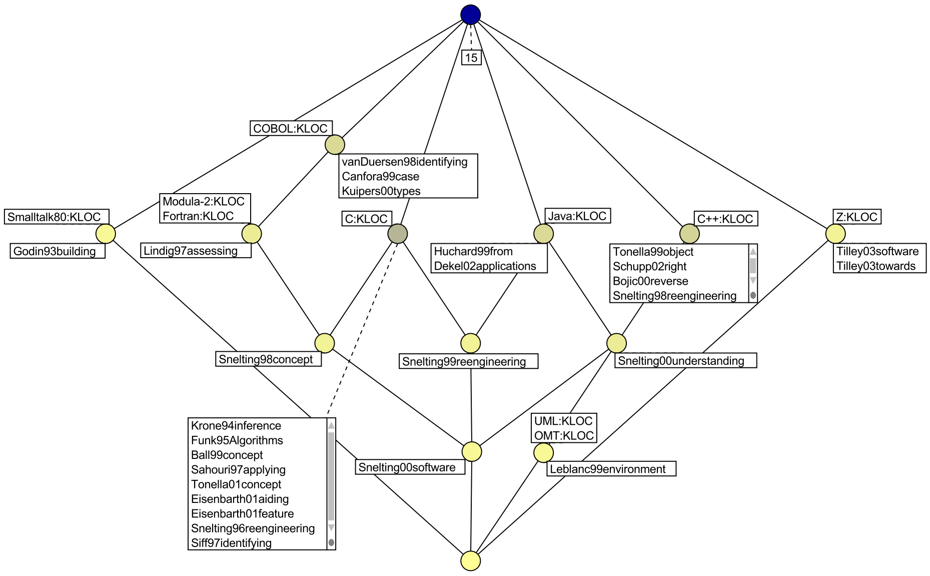

Fig. 2. Lattice based on Table 1 showing reported application languages ignoring size.

A summary of collaboration between the authors within the survey papers is presented in Figure 4. This concept lattice represents those authors who have collaborated on papers with different authors. There are 13 papers at the top of the lattice whose authors only appear once across the 42 papers or who always work with the same co-authors. Only a count of the number of papers is shown and the structures below Godin, Boettger and Snelting are of particular interest. These structures represent collaboration on multiple papers with a reasonably large number of different authors. Snelting's collaboration with others is the likely reason for the large number of language applications described in *Snelting00software*.

Finally in this section, Figure 5 presents a line diagram showing the transitive closure of citations within the set of survey papers. For example, if paper B cites paper C, and paper A cites paper B then A transitively cites C.

It can be observed that most of the work is nearly linearly in terms of citations reflecting coherence within the community. For example Snelting and Godin cite each others work before large forks appear in the structure. Papers with the most "impact" appear at the top of the diagram with long chains beneath them. The 17 papers at the bottom of the diagram are uncited within the survey literature.

Note that only 38 of the 42 survey papers appear in this diagram. The papers: *Eisenbarth01feature, Tilley03software, Tilley03towards* and *Tonella01concept* were not

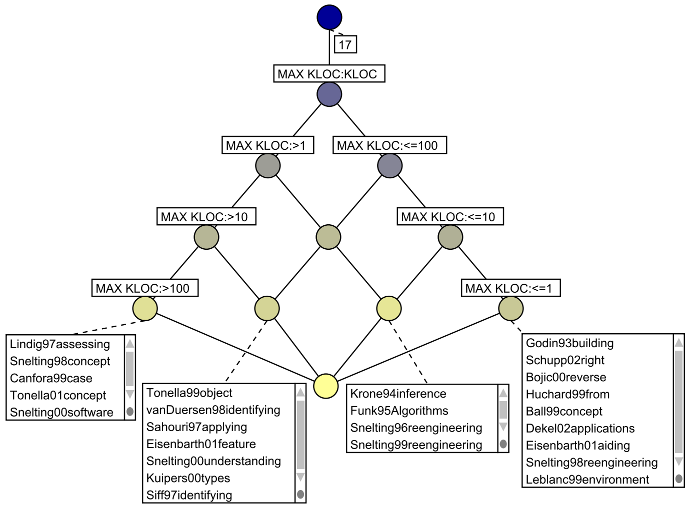

Fig. 3. An inter-ordinal scale showing the maximum application size across all languages in each paper. Note that papers reporting UML, OMT and Z are ignored in this scale because KLOC is not a suitable metric for design/specification languages.

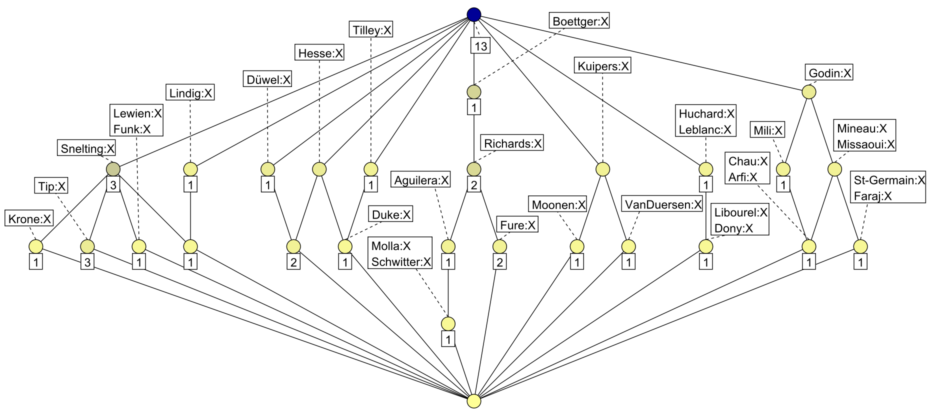

Fig. 4. Lattice showing collaboration between authors within the set of survey papers. Note that only papers where the authors have worked with different co-authors appear.

available when the citation closure was computed. In addition, complete references for *Andelfinger97diskursive* and *Boettger01modelling* were not available to the authors at the time of publication.

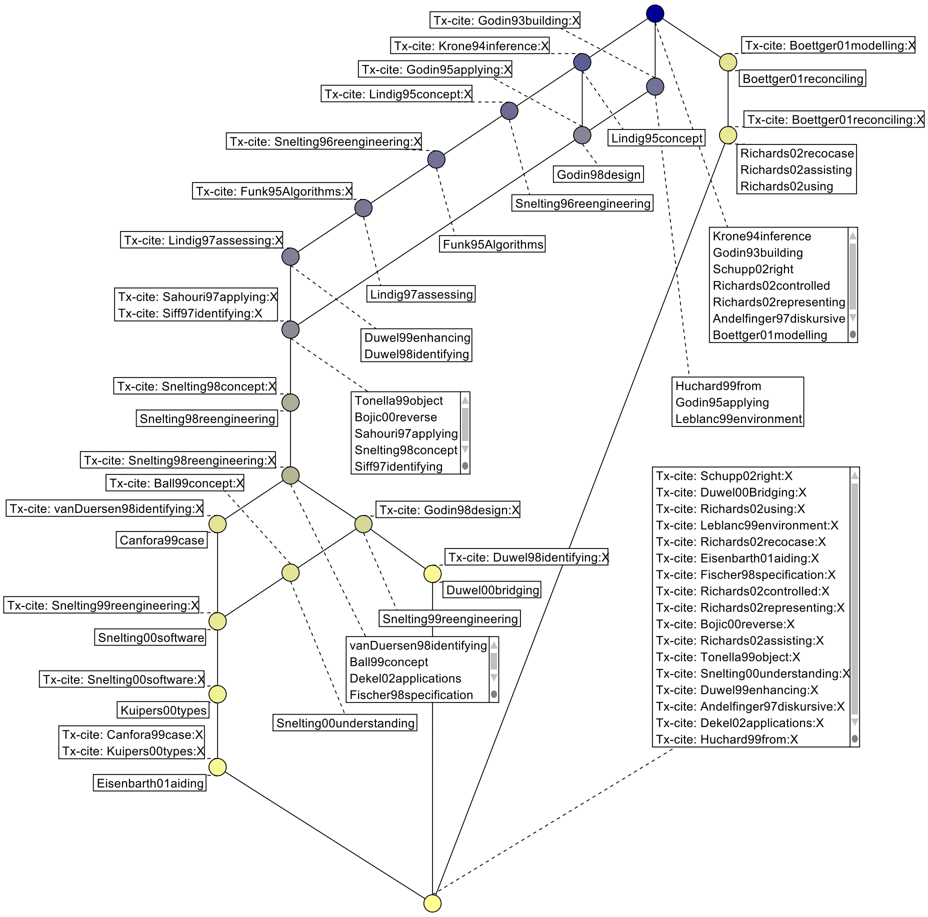

Fig. 5. Concept lattice representing the transitive closure of citations within the set of survey papers.

4 Early Phase Activities

Software engineering activities that occur before the commencement of coding can be considered as “early-phase” activities. This section summarises the papers shown in Figure 1 that support the early-phase software engineering activities: requirements analysis, architectural design and detailed design. An overview of papers describing FCA support for software maintenance activities appears in Section 5

4.1 Requirements Analysis

Andelfinger's thesis[1] describes a discoursive environment for requirements gathering based on Habermas' theory of "communicative rationality". Habermas described a perhaps somewhat idealistic discoursive environment that attempts to make any agendas obvious during negotiation and considers all viewpoints equally.

Within the thesis FCA is used as a question answering and discussion tool. The value of unrealised concepts is highlighted as it promotes questions about things that are missing which may be indicative of incomplete requirements. While the three case studies presented are not directly related to software engineering they parallel typical problems encountered in requirements gathering. It is interesting to note that one of the case studies (Beta) describes the gathering requirements for an FCA based retrieval system for the library of the "Center of Interdisciplinary Technology Research"(ZIT)[41].

Use Cases are a tool used in requirements gathering and analysis where a task is described from a certain perspective or role. Typically these descriptions are written in natural language although sometimes controlled vocabularies are used. The work of Düwel [14] and Düwel and Hesse [16,15] attempts to identify class candidates in use case descriptions. The use cases themselves are considered as objects in a formal context and nouns identified within the text are considered as formal attributes. The structure of the corresponding concept lattice is then considered as a starting point for a class hierarchy. In [58] Tilley et al. present a case study applying Düwel's approach to an Object-Z specification.

Böttger and Richards et al.[37,36,39,7,35,38,40] also apply FCA to use cases in an attempt to reconcile descriptions written by different stake-holders using a controlled vocabulary and grammar. Their tool RECOCASE (RECOnciling CASE tool) exploits a Prolog answering system called ExtrAns and LinkGrammar — an English language parser using link grammar theory — to translate sentences into unambiguous flat logical forms. The formal nature of this controlled language facilitates the analysis of use cases to identify misunderstandings, inconsistencies and conflicts. Furthermore, a context can be produced where the sentences are formal objects and the flat logical forms are broken into word phrases which are then treated as formal attributes. Similar concepts and differences in terminology can then be identified from the resulting concept lattice.

4.2 Component Retrieval for Software Reuse

Lindig[33] describes a retrieval system that could be used for retrieving software components from a library indexed by keywords. An example using keywords from man pages describing Unix commands is presented where the keywords are considered as formal attributes and the Unix commands as formal objects. In this case the commands represent the components to be retrieved. The retrieval system provides a query by refinement interface in which a boolean query, B to (B', B''), and the concepts in the lower cover are offered as refinements to the user.

4.3 Formal Specification

Fischer[19] builds on the component retrieval work of Lindig, however, instead of keywords a formal specification that captures the behaviour of a software component is used. By exploiting the power of Formal Methods components can be retrieved based on explicit properties required for a components selection or on implicit similarity with other components. The underlying structure is a formal concept lattice which is computed "off-line" and it facilitates the browsing of software component libraries as well as retrieval.

The component specifications consist of axioms describing pre-conditions and post-conditions. These specifications are used as both the objects and the attributes of a formal context. Functions or partial functions within the specifications are also considered as formal attributes. In addition to library navigation and component retrieval, the resulting lattice can also be used to improve the library. Unlabelled concepts and extents containing intuitively "unexpected" components may indicate missing attributes or features that can then be added to the library.

4.4 Visualizing Z Specifications via FCA

Z is a state based formal method that exploits set theory and first order predicate logic. Specifications in Z are composed of named *schema* boxes that describe operations by their input/output behaviour. Models are constructed by specifying and composing a series of schemas which can be further refined to reflect the desired level of system abstraction. An example schema from Spivey's "BirthdayBook" specification[55] appears in Figure 6.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddBirthday}} \\
\hline
\Delta \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{date}? : \textit{DATE} \\
\hline
\textit{name}? \notin \textit{known} \\
\textit{birthday}' = \textit{birthday} \cup \{\textit{name}? \mapsto \textit{date}?\} \\
\hline
\end{array}
$$

Fig. 6. The AddBirthday schema from the BirthdayBook specification in Z.

As a result of the mathematical nature of the notation most Z tools are comprised of at least a formatting package for LaTeX and a type-checker. There is a continued call for formal methods tool support and in particular tools that support the ability to view specifications at different levels of abstraction. This section provides a brief overview of the authors work to create a tool for interactively exploring Z specifications based on ZML[56], an XML representation of Z, and the open-source, cross-platform FCA tool ToscanaJ. This work has recently been reported in [57] and an overview of the process is presented in Figure 7.

While Z specifications are traditionally written in LaTeX and viewed as postscript or PDF documents, a more recent alternative for "marking-up" specifications is ZML. Specifications written in ZML are easily parsed and have the advantage that they can be transformed using XSL stylesheets into HTML which can be rendered in a Web

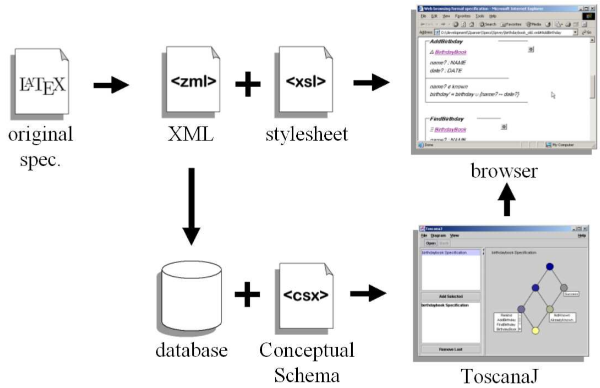

Fig. 7. Overview of the specification browsing system.

browser. Specifications can then be easily accessed in an on-line form complete with HTML anchors and hyperlinks for navigation.

ToscanaJ is a FCA tool for rendering concept lattices that supports extent/intent highlighting, conceptual scaling, nesting, and zooming[11]. A formal context representing the static structure of a Z specification can be constructed by considering each schema as a formal object and the individual mark-up elements as attributes. This context can be stored in a relational database and accessed using ToscanaJ. Scales can either be constructed by the user or pre-defined to "query" the specification and reveal properties of interest based on, for example, Z language features or data-types within a specification. ToscanaJ's extensible view interface can then be exploited so users can click on a schema name within a line-diagram and a web browser will be launched displaying the relevant part of the specification rendered using ZML. In this way users can conceptually navigate and explore a Z specification using FCA and retrieve the relevant parts of the original specification as desired.

5 Software Maintenance

While Z specifications typically describe systems being developed, this section deals with the use of FCA to suggest modifications to existing software programs or systems. This activity may be performed within different contexts and with different objectives. The term software maintenance usually refers to the modification of a software system that has already been deployed to the customer. Four types of software maintenance are identified

1. corrective — modifying a system to improve the way it meets its requirements.
2. adaptive — modifying a system to operate correctly in a new environment
3. perfective — adding new functionality to the system
4. preventative — improving the design and implementation to better accommodate future maintenance activities

The spectrum of application of FCA to software maintenance cover each of these types, but have a common thread — extracting understandable structures that organise the artifacts of software systems. The various applications of FCA to software maintenance vary on their inputs, the concept lattices they create, and the use to which they put the concept lattices. In an effort to organise and present these different approaches they are grouped into the following categories: analysis of dynamic information, application to legacy systems, and review of class hierarchies. Following this we will present a framework for merging these approaches using a knowledge base storing artifacts of the software system and relationships between them and a mechanism for deriving concept lattices using graph based queries.

5.1 Dynamic Analysis

Approaches for the analysis of dynamic aspects of software systems have been reported by Ball[2], Eisenbarth et al.[17,18], and Bojic et al.[5]. Ball examines test coverage while Bojic and Eisenbarth recover software architecture related to use cases.

Dynamic information is typically extracted from programs by executing them with a profiler. The profiler records which software artifacts such as procedures and variables were accessed during the run of the program and in what order..

A computer program essentially consists of a large number of instructions. Each instruction is identified by its position within the program. For example we may consider the instruction at memory location `0x00f1`. A run of a computer program produces a *trace*. A trace is the sequence of instructions that were run. Two notions concerning instructions, with respect to a collection of traces, are important: dominance and predomination. An instruction x *dominates* another instruction y if any trace prefix that ends on y contains x. In other words x dominates y if the only way to execute y is to have already executed x. Similarly x *post-dominates* y if any trace postfix starting with y also contains x. In other words x post-dominates y if any execution of y indicates that x will subsequently be executed.

While there is not a direct correspondence between dominance, post-dominance and the lattice structure, dominance and post-dominance do lead to implications in the lattice. If we consider program traces as formal objects, and instructions as the formal attributes then if x either dominates or post-dominates y then there will be an implication in the lattice that $y \to x$.

Ball examines test coverage by comparing the implicational logic in the concept lattice generated from traces extracted from test programs with dominance and post-dominance relationships extracted by static code analysers. Any additional implications in the concept lattice are considered to see if they can be removed by the introduction of a new test.

Eisenbarth et al. report an experiment attempting to identify parts of the software structure of two web browsers related to use cases. For example, they are interested in locating the portion of the web-browsers related to use of the browser history. They construct a number of test cases (scenarios) which cover the use cases of interest. A concept lattice is produced with scenarios as formal objects and program artifacts such as methods, and variables as formal attributes. The attribute contingents of object concepts are of particular interest since they contain the program artifacts introduced by that particular scenario.

Bojic et al. report a similar approach but additionally the artifacts within the attribute contingents are arranged as UML diagrams using a UML reverse engineering tool. In this way the specific parts of the software architecture related to use cases can be extracted and viewed. The capability is particularly useful in the preparation of traceability in the software engineering process whereby aspects of the system architecture can be traced back to requirements.

5.2 Application to legacy systems

FCA has been applied to extract structure from legacy systems and has been compared with hierarchical clustering as a technique for organising the artifacts of legacy systems.

Snelting[49,30,21] used FCA to analyses the preprocessor commands in legacy C programs in order to examine the configuration structure. Formal objects are the code fragments included by the preprocessor commands, while, the formal attributes are disjunctive expressions governing the inclusion of the code fragments. The concept lattice is constructed and the notion of an interference is introduced. An *interference* is a meet reducible concept with a non empty extent. Two types of undesirable interference are identified, those corresponding to illegal configurations — for example an interference between XWINDOWS and DOS — and those corresponding to orthogonal attributes — for example an interference between a variable related to the graphics subsystem and one related to the operating system.

In order to make the concept lattices produced more manageable *horizontal decompositions* are introduced[21]. A horizontal sum is defined as:

$$\sum_{i=1}^{n} L_i = \{\top, \bot\} \cup \bigcup L_i \; \{\top_i, \bot_i\}$$

The constituent elements of the sum L_i are usually disjoint. Experiments with legacy systems revealed that few configuration lattices can be directly decomposed into a horizontal sum of disjoint sub-lattices. In order to simplify the configuration structure the notion of a k-*interference* is introduced. A k-interference is a collection of k meet reducible incomparable concepts whose downset removal yields a decomposition into a disjoint horizontal sum.

The concepts involved in such k-interferences are of particular interest since they are most likely interferences between orthogonal aspects of the system configuration.

Other techniques to simplify the concept lattice include limiting the nesting depth of preprocessor commands considered and merging rows which differ by fewer than k elements. These techniques are of use when the objective is to get an overview of the configuration structure present in a software program.

Legacy programs written in languages where access to common data structures is the norm, e.g. FORTRAN and COBOL, have been considered by van Deursen et al.[61], Kuipers et al.[31], Lindig et al.[34] and Canfora[9].

Van Deursen and Kuipers compare the use of formal concept analysis for grouping fields within a large legacy COBOL program to that of hierarchical clustering. Considered Hierarchical clustering involved defining a distance metric between COBOL procedures, extending the metric to sets of procedures, starting with every procedure in its own cluster and then repeatedly merging the two closest clusters to produce a binary tree of clusters. Hierarchical clustering is generally criticised because it can yield different inputs for the same data as in some instances several clusters are equidistant and different results for slightly different distance metrics are obtained. The results produced in contrast by FCA are always the same, not dependent on the definition of a distance metric, and were much closer to that produced by software engineers familiar with the legacy system. Since the objective was to focus on domain specific procedures rather than those performing system functions, procedures having a high degree of fan-in were judged as being system procedures and were discarded. This judgement was controlled by an operator set threshold.

Canfora et al. follow a similar approach but are interested in organising a legacy COBOL system into components suitable for distribution via CORBA. They considered programs and their use of files (relational tables). The formal context was pruned by removing objects and attributes in isolated concepts — concepts that are directly below the top concept and directly below the bottom concept and therefore don't have any intent or extent intersection with other concepts[6]. Files, i.e. relation tables, having the same structure were also merged. This case arises when several files are used to perform some operation on a table — like sorting it. Also programs that used only a single file were removed. Canfora et al. apply their rules until no more formal objects and formal attributes can be removed. The result was a concept lattice that was almost horizontally decomposable — in the sense of Snelting et al. — into four domain areas, except for a number of interferences corresponding to operations involving more than one domain area.

The task of deriving object oriented models from legacy systems written in C has also been considered by Sahouri et al.[44], Siff and Reps[48], and Tonella[59]. The general approach is to consider C functions as formal objects and the attributes as either commonly accessed data structures or fields within commonly used structures.

Both Siff and Reps, and Tonella are concerned with re-organsing the functions into a different, perhaps more fine grained, module structure based on the access of functions to either (i) common data structures[59], or (ii) fields within commonly accessed data types[48].

In Tonella's approach a modular structure is obtained by selecting concepts with pairwise disjoint extents, assigning formal attributes to each selected concept, and measuring (i) the number of selected concepts and, (ii) the number of objects that access formal attributes not assigned to their concept. Formal objects not having any attributes must be discarded. A process is performed whereby the the two measures are optimised.

5.3 Reengineering Class Hierarchies

Snelting[52,51,54] explains a mechanism to re-organise class hierarchies using FCA. Variables in C++ are taken as formal objects and methods and fields of the objects to which the variables refer are taken as formal attributes. A variables is associated with a field or method if that variable is used to access the method or field. A number of rules

[6] other concepts except for the top and bottom concepts

are employed to account for assignment between variables and conservatively account for dynamic dispatch. The real objective of investigation is the objects existing during a run of a program and Snelting and Tip access these using static analysis and via the medium of variables.

Schupp et al. consider class hierarchies in the C++standard template library (STL). They have classes as formal objects and documented properties of the classes as formal attributes. They introduce the notions of "well abstracting", "lacking orthogonality" and "lacking refinement" to describe class libraries. Rather than inspecting various aspects of the structure they attempt to construct the whole concept lattice, render it and draw conclusions. Inspection of aspects of the STL as shown, for example, in Figure 8 reveal a very regular structure. The example shows three complementary pairs of attributes: unique and multiple associative, sorted and hashed, and pair and simple associative. Complementary attributes are related by *exclusive or* — in other words all objects have exactly one of the two attributes.

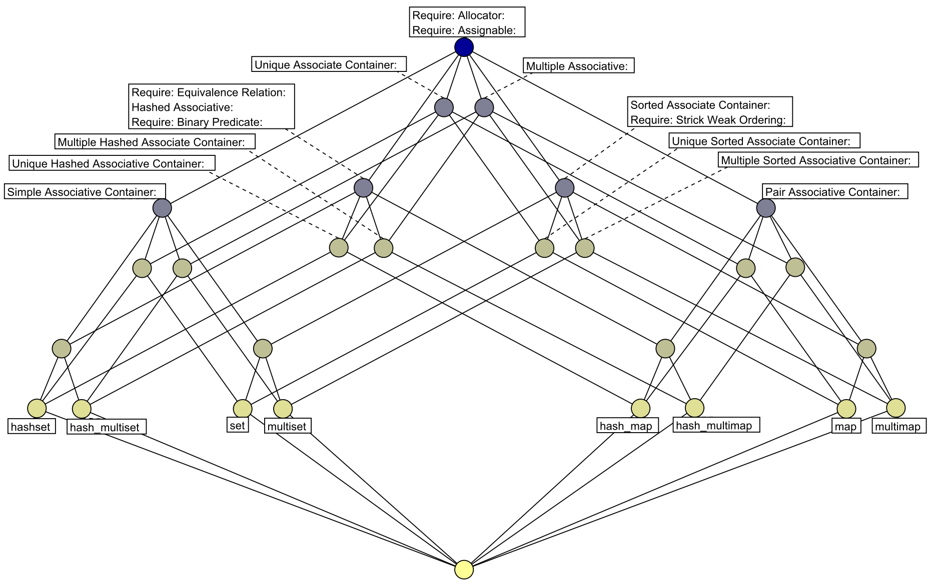

Fig. 8. Concept lattice of set and map classes in the STL.

Godin et al.[23] consider a context where the formal objects are messages (methods in SMALLTALK) and formal attributes are classes. They consider concepts having an empty attribute contingent, i.e. those not labelled by a class as new class candidates. We applied a similar technique to an analysis of the collection classes in JAVA and discovered a new inheritance link not present in the collection. Our approach is briefly outlined in Section 5.4. Godin et al.[24] further incorporate static call graph information into the concept lattice.

Huchard et al.[27] consider a concept lattice generated with classes as formal objects and attributes derived from method signatures, thus including information about parameter types and return values. Again each concept is considered as a candidate for a JAVA interface.

Tonella et al.[60] attempt to recover design patterns in source code using a context in which the formal objects are n-tuples (in practice triples are used) whose elements are types in the software, and formal attributes are triples of the form (i, j, r) where i and j are indexes into the n-tuple and r is a relation type. So for example an object, (A, B, C), being associated with an attribute $(1, 2, derived - from)$ would indicate that A is derived from B. Tonella et al. discover as one of the concepts in the concept lattice the well known "adapter pattern".

5.4 Conceptual Analysis of Software Structure

The process of software design and implementation often contains many arbitrary decisions, such as the name of methods or variables or indeed sometimes the structure of a class hierarchy. As the design proceeds these decisions need to be reviewed in order to achieve consistent, orthogonal and simple designs. Agile methods, and extreme programming (XP) in particular, advocate regular refactoring activities undertaken to regularise and revise the software structure[20,3]. Our tool, Conceptual Analysis of Software Structure (CASS)[10], attempts to address these requirements. CASS exploits source code analysers and profilers to extract information which is stored in a knowledge base as a large collection of triples of the form *(subject, predicate, object)*. A rule based system is then used to extend the knowledge base with new relationships and artifacts. Graph based queries can then be used to define an aspect of the code to be explored and these are used to generate result sets that are visualised using concept lattices. These hypotheses or questions may then be investigated either by generating new lattices, perhaps displaying new aspects of the software structure, or by navigating back to the source artifacts within the software or its documentation. Since each concept lattice is generated from a query graph, a natural refinement ordering allows general views to be elaborated and made more specific. Thus the user is able to progress from a general view to a more specific view, or vice versa. In addition, the theory of formal concept analysis (FCA) allows two or more aspects of the software structure to be combined coherently in nested diagrams.

6 Conclusion

This paper has provided a snapshot of where FCA has been applied to support software engineering activities using the activities defined in the ISO 12207 standard and a number of other criteria. The power of a line-diagram to convey and summarise large amounts of information has been demonstrated on real-world examples via a number of approaches and tools. The paper has also provided an insight into authorship groups and citation patterns within the survey literature.

The majority of the reported work has been in the areas of detailed design and software maintenance where FCA has been applied to OO re-engineering and class identification tasks. While these approaches could be seen as obvious applications because of the specialisation/generalisation relationship between the concepts in a formal concept lattice, the range of different formal objects ranging from compiled code through to use-cases is surprising. Other novel applications have included support for test-coverage analysis and perhaps in the future tools and examples will emerge that support coding, integration and qualification testing.

References

1. U. Andelfinger. *Diskursive Anforderungsanalyse. Ein Beitrag zum Reduktionsproblem bei Systementwicklungen in der Informatik*. Peter Lang, Frankfurt, 1997.
2. T. Ball. The concept of dynamic analysis. In *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 216–234, September 1999.
3. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
4. B.W. Boehm. A spiral model of software development and enhancement. In R.H. Thayer, editor, *Tutorial: Software Engineering Project Management*, pages 128–142. IEEE Computer Society, Washington, 1987.
5. D. Bojic and D. Velasevic. Reverse engineering of use case realizations in uml. In *SAC2000*. ACM, 2000. `http://www.acm.org/conferences/sac/sac00/Proceed/FinalPapers`.
6. K. Böttger. Modelling and reconciling functional requirements from differenet viewpoints using use case/scenarios and formal concept analysis. Master's thesis, University of Mannheim, Germany, 2001.
7. K. Böttger, R. Schwitter, D. Richards, O. Aguilera, and D. Mollá. Reconciling use cases via controlled language and graphical models. In *INAP'2001 - Proceedings of the 14th International Conference on Applications of Prolog*, pages 20–22, Japan, October 2001. University of Tokyo.
8. P. Burmeister. Formal concept analysis with ConImp: Introduction to the basic features. Technical report, TU-Darmstadt, Darmstadt, Germany, 1996. `http://www.mathematik.tu-darmstadt.de/~burmeister/`.
9. G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca. A case study of applying an eclectic approach to identify objects in code. In *Workshop on Program Comprehension*, pages 136–143. IEEE, 1999.
10. R. Cole and T. Tilley. Conceptual analysis of software structure. In *Proceedings of 15th International Conference on Software Engineering and Knowledge Engineering, SEKE'03*, July 2003. In Press.
11. R.J. Cole and P.W. Eklund. Browsing semi-structured web texts using formal concept analysis. In *Proceedings 9th International Conference on Conceptual Structures*, LNAI 2120, pages 319–332, Berlin, 2001. Springer-Verlag.
12. U. Dekel. Applications of concept lattices to code inspection and review. In *The Israeli Workshop on Programming Languages and Development Environments*, chapter 6. IBM Haifa Research Lab, IBM HRL, Haifa University, Israel, July 2002. `http://www.haifa.il.ibm.com/info/ple/papers/inspec_summary.pdf`.
13. H. Dicky, C. Dony, M. Huchard, and T. Libourel. ARES, adding a class and restructuring inhertitance hierarchy. In *BDA : Onzimes Journes Bases de Donnes Avances*, pages 25–42, 1995.
14. S. Düwel. Enhancing system analysis by means of formal concept analysis. In *Conference on Advanced Information Systems Engineering 6th Doctoral Consortium*, Heidelberg, Germany, June 1999.

15. S. Düwel and W. Hesse. Identifying candidate objects during system analysis. In *Proceedings of CAiSE'98/IFIP 8.1 3rd International Workshop on Evaluation of Modelling Methods in System Analysis and Design (EMMSAD'98)*, Pisa, 1998.
16. S. Düwel and W. Hesse. Bridging the gap between use case analysis and class structure design by formal concept analysis. In J. Ebert and U. Frank, editors, *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Proceedings "Modellierung 2000"*, pages 27–40, Koblenz, 2000. Fölbach-Verlag.
17. T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings ICSM*. IEEE Computer Society Press, 2001.
18. T. Eisenbarth, R. Koschke, and D. Simon. Feature-driven program understanding using concept analysis of execution traces. In *9th International Workshop on Program Comprehension*, pages 300–309. IEEE, 2001.
19. B. Fischer. Specification-based browsing of software component libraries. In *Automated Software Engineering*, pages 74–83, 1998.
20. M. Fowler. *Refactoring, Improving the Design of Existing Code*. Addison Wesley, 1999.
21. P. Funk, A. Lewien, and G. Snelting. Algorithms for concept lattice decomposition and their application. Technical Report 95-09, Computer Science Dept., Technische Universitat Braunschweig, 1995. `http://citeseer.nj.nec.com/117356.html`.
22. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.
23. R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of the OOPSLA'93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 394–410, 1993.
24. R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of class hierarchies based on concept (galois) lattices. *Theory and Application of Object Systems (TAPOS)*, 4(2):117–134, 1998.
25. R. Godin, G. Mineau, R. Missaoui, M. St-Germain, and N. Faraj. Applying concept formation methods to software reuse. *International Journal of Knowledge Engineering and Software Engineering*, 5(1):119–142, 1995.
26. B. Groh, S. Strahringer, and R. Wille. Toscana-systems based on thesauri. In *Proceedings of the 6th International Conference on Conceptual Structures*, LNAI 1453, pages 127–138, Berlin, 1998. Springer Verlag.
27. M. Huchard and H. Leblanc. From java classes to java interfaces through galois lattices. In *Actes de ORDAL'99: 3rd International Conference on Orders, Algorithms and Applications*, pages 211–216, Montpellier, 1999.
28. IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, New York, September 1990.
29. IEEE. *Standard for Info. Technology – Software life cycle processes*. IEEE, New York, March 1998.
30. M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proceedings of ICSE 1994*, pages 49–57, 1994.
31. T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. Technical Report SEN-R0017, Centrum voor Wiskunde en Informatica, July 2000.
32. H. Leblanc, C. Dony, M. Huchard, and T. Libourel. An environment for building and maintaining class hierarchies. In A. Moreira and S. Demeyer, editors, *ECOOP'99: Workshop "Object-Oriented Architectural Evolution"*, LNCS 1743. Springer-Verlag, 1999.
33. C. Lindig. Concept-based component retrieval. In J. Köhler, F. Giunchiglia, C. Green, and C. Walther, editors, *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, pages 21–25, August 1995.

34. C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the International Conference on Software Engineering (ICSE 97)*, pages 349–359, Boston, 1997.
35. D. Richards and K. Boettger. Assisting decision making in requirements reconciliation. In *Proceedings of the 7th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2002)*, Brazil, September 2002.
36. D. Richards and K. Boettger. A controlled language to assist conversion of use case descriptions into concept lattices. In *Proceedings of 15th Australian Joint Conference on Artificial Intelligence*, 2002.
37. D. Richards and K. Boettger. Representing requirements in natural language as concept lattices. In *22nd Annual International Conference of the British Computer Society's Specialist Group on Artificial Intelligence (SGES), (ES2002)*, Cambridge, December 2002.
38. D. Richards, K. Boettger, and A. Fure. RECOCASE-tool: A CASE tool for RECOnciling requirements viewpoints. In *Proceedings of the 7th Australian Workshop on Requirements Engineering, AWRE'2002*, 2002.
39. D. Richards, K. Boettger, and A. Fure. Using RECOCASE to compare use cases from multiple viewpoints. In *Proceedings of the 13th Australasian Conference on Information Systems ACIS 2002*, Melbourne, December 2002.
40. D. Richards and P. Compton. Combining formal concept analysis and ripple down rules to support reuse. In *Proceedings of Software Engineering Knowledge Engineering SEKE'97*, Madrid, June 1997. Springer-Verlag.
41. T. Rock and R. Wille. Ein TOSCANA — erkundungsystem zur literatursuche. In G. Stumme and R. Wille, editors, *Begriffliche Wissensverabeitung: Methoden und Anwendungen*, pages 239–253, Berlin-Heidelberg, 2000. Springer-Verlag.
42. P. Rook. Controlling software projects. *Software Engineering Journal*, 1(1):7–16, January 1996.
43. W. W. Royce. Managing the development of large software systems. In R.H. Thayer, editor, *Tutorial: Software Engineering Project Management*, pages 118–127. IEEE Computer Society, Washington, 1987. Originally published in Proceedings of WESCON'97.
44. H.A. Sahraoui, W. Melo, H. Lounis, and F. Dumont. Applying concept formation methods to object identification in procedural code. In *Proceedings of International Conference on Automated Software Engineering (ASE '97)*, pages 210–218. IEEE, November 1997.
45. I. Schmitt and S. Conrad. Restructuring object-oriented database schemata by concept analysis. In T. Polle, T. Ripke, and K.-D. Schewe, editors, *Fundamentals of Information Systems (Post-Proceedings 7th International Workshop on Foundations of Models and Languages for Data and Objects FoMLaDO'98)*, pages 177–185, Boston, 1999. Kluwer Academic Publishers.
46. I. Schmitt and G. Saake. Merging inheritance hierarchies for database integration. In *Proceedings of the 3rd International Conference on Cooperative Info. Systems (CoopIS'98)*, New York, August 1998.
47. S. Schupp, M. Krishnamoorthy, M. Zalewski, and J. Kilbride. The "right" level of abstraction—assessing reusable software with formal concept analysis. In G. Angelova, D. Corbett, and U. Priss, editors, *Foundations and Applications of Conceptual Structures - Contributions to ICCS 2002*, pages 74–91. Bulgarian Academy of Sciences, 2002.
48. M. Siff and T. Reps. Identifying modules via concept analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.
49. G. Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

50. G. Snelting. Concept analysis - a new framework for program understanding. In *SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 1–10, Montreal, Canada, June 1998.
51. G. Snelting. Software reengineering based on concept lattices. In *Proceedings 4th European Conference on Software Maintenance and Reengineeering*, pages 3–12. IEEE, 2000.
52. G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. Technical Report RC 21164(94592)24APR97, IBM T.J. Watson Research Center, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA, 1997.
53. G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of ACMSIGSOFT Symposium on the Foundations of Software Engineering*, pages 99–110, November 1998.
54. G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Trans. on Programming Languages and Systems*, pages 540–582, May 2000.
55. J.M. Spivey. *The Z notation : a reference manual*. Prentice-Hall International, 1989.
56. J. Sun, J.S. Dong, J. Lui, and H. Wang. Object-Z web environment and projections to UML. In *WWW10 - 10th International World Wide Web Conference*, pages 725–734, New York, 2001. ACM.
57. T. Tilley. Towards an FCA based tool for visualising formal specifications. In *Contributions to ICCS2003 - the 11th International Conference on Conceptual Structures*, Dresden, July 2003. In Press.
58. T. Tilley, W. Hesse, and R. Duke. A software modelling exercise using FCA. In *Contributions to ICCS2003 - the 11th International Conference on Conceptual Structures*, Dresden, July 2003. In Press.
59. P. Tonella. Concept analysis for module restructuring. *IEEE Trans on Software Engineering*, 27(4):351–363, April 2001.
60. P. Tonella and G. Antoniol. Object-oriented design pattern inference. In *Proceedings of CSM 1999*, pages 230–240, 1999.
61. A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of 21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.